The Thesis Committee for Haiyan Z. Fateh
Certifies that this is the approved version of the following thesis:

Modeling and Measurements of Thermoelectric Waste Heat Recovery Devices for Motor Vehicles

APPROVED BY
SUPERVISING COMMITTEE:

Matthew Hall

Li Shi

Modeling and Measurements of Thermoelectric Waste Heat Recovery Devices for Motor Vehicles

by

Haiyan Z. Fateh, B.S.

Thesis

Presented to the Faculty of the Graduate School of
The University of Texas at Austin
in Partial Fulfillment
of the requirements
for the degree of

Master of Science in Engineering

The University of Texas at Austin
December 2013

Dedication

To Mom, Dad, and my family for encouraging me to do my best in my education.

Acknowledgements

This study could not have been carried out without the help of many bright individuals I met along the way. I would like to thank my advisor, Dr. Hall, for his extensive guidance on my modeling and experimental analysis of thermoelectric generators. Dr. Hall is one of the most understanding and humble people I have met and I have utmost respect for him. I owe him huge gratitude for listening to my ideas, always being available to speak to me, and for providing constructive remarks in my study. His enthusiasm in hands on experimental works proved to be extremely important to me as he was never afraid to join me in my experiments and troubleshoot problems. His excellent professional attitude was an ideal example for me.

I would also like to thank Dr. Shi for his guidance throughout my graduate school. His assessment and critique of my work helped me realize the mistakes I made along the way. His thorough analysis of my work helped me learn from my mistakes, move in the right direction, and draw appropriate conclusions from my work. Dr. Zhou has also been of great help to me during my experimental work related to thermoelectric materials. I would like to thank him for his guidance during thermoelectric material synthesis. His knowledge and experience in experimental techniques regarding materials and processes was an avenue for me to learn new skills and techniques from.

Chad Baker was a valuable friend and a colleague whose efforts in introducing me to this project and helping me learn various numerical modeling methods were instrumental in making this study possible. He was honest and direct with his advice which helped me make some important decisions during my early days in graduate school. I would also like to thank Libin Zhang and Xi Chen. Their materials science background was extremely valuable considering the nature of this study. They were always available to work with me during material synthesis experiments.

Abstract

Modeling and Measurements of Thermoelectric Waste Heat Recovery Devices for Motor Vehicles

Haiyan Z. Fateh, M.S.E.

Supervisor: Matthew Hall

This study is centered on modeling and experimental efforts to simulate and optimize the performance of thermoelectric generators (TEGs) for waste heat recovery systems for use in motor vehicles. TEGs are being studied and developed for applications in which waste heat, for example, from the exhaust of motor vehicles is converted into usable electricity. TEGs consisting of TE elements integrated with an exhaust heat exchanger require optimization to produce the maximum possible power output. Important optimization parameters include TE element leg length, fill fraction, leg area ratio between n- and p-type legs, and load resistance. A finite difference model was developed to study the interdependencies among these optimization parameters for thermoelectric elements integrated with an exhaust gas heat exchanger. The present study was carried out for TE devices made from n-type Mg_2Si and p-type $MnSi_{1.8}$ based silicides, which are promising TE materials for use at high temperatures associated with some exhaust heat recovery systems. The model uses specified convection boundary conditions instead of specified temperature boundary conditions to duplicate realistic operating conditions for a waste heat recovery system installed in the exhaust of a vehicle. A numerical model for a new waste heat recovery system configuration was proposed which showed an improvement of 40% in net power output over the conventional systems while using approximately 60% more TEG modules. The 1st

generation, and an improved 2nd generation TEG module using n-type Mg_2Si and p-type $MnSi_{1.8}$ based silicides were fabricated and tested to compare and correlate TE power generation with the numerical model. Important results include parameter values for maximum power output per unit area and the interdependencies among those parameters. Heat transfer through the void areas was neglected in the numerical model. When thermal contact resistance between the TE element and the heat exchangers is considered negligible, the numerical model predicts that any volume of TE material can produce the same power per unit area, given the parameters are accurately optimized. Incorporating the thermal contact resistance, the numerical model predicts that the peak power output is greater for longer TE elements with larger leg areas. The optimization results present strategies to improve the performance of TEG modules used for waste heat recovery systems.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

1.1 Motivation

Energy conservation is becoming increasingly important because of growing energy demands. This compels us to promote the development of energy efficient systems. For instance, motor vehicles are subject to EPA standards for exhaust emissions, and fuel economy. One of the main concerns with burning fossil fuels in motor vehicles is the emission of greenhouse gases. In the past, CO_2 has been responsible for almost 80% of the gaseous emissions [1], whose negative environmental impacts and effects on global warming are well-documented in literature [2,3,4,5,6,7,8].

Exhaust gases from the tail pipes of motor vehicles are responsible for dissipating roughly one third of the energy content of the fuel to the environment as heat [9,10,11]. Incorporating waste heat recovery technologies in vehicles can lead to increased fuel economy and therefore contribute to lesser emissions. Hence, various waste heat recovery technologies are being investigated to capture that waste heat. Such technologies include organic Rankine cycles [12], turbo-compounding [13], direct use of the waste heat as thermal energy, power absorption chillers [14], load preheating, regenerative burners [15], and thermoelectric devices [12,16,17,18,19], among others. Thermoelectric devices in particular have gathered considerable attention in the last few decades. They are solid state heat engines that use the Seebeck effect to directly convert heat to electrical energy. It has been concluded that TEG devices installed in the exhaust of a vehicle are applicable to typical mid-sized motor vehicles [20]. Furthermore, they operate quietly without moving parts [21], and they are compact and lightweight. They do not produce any emissions either [22]. However, thermoelectric devices have low efficiency (<5%) in addition to being expensive; this has limited their widespread use. Nonetheless, potential use of relatively inexpensive materials like n-type Mg_2Si and p-type $MnSi_{1.8}$ based silicides shows promise for cost reduction and improved performance [23].

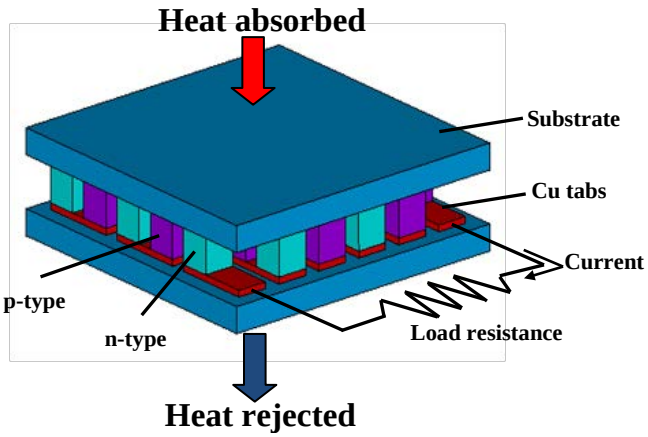


Fig. 1: Schematic of a TEG module

A schematic of a typical TEG (Thermoelectric Generator) module is shown in Fig. 1. System level thermoelectric heat exchangers for waste heat recovery consist of several TEG modules integrated with a compact heat exchanger that can extract maximum heat from the exhaust flow.

1.2 Literature Review

Substantial research has been done in recent years regarding the design and optimization of thermoelectric heat exchangers. Applications of various thermoelectric materials exhibiting peak thermoelectric efficiencies at different temperature ranges have been studied in the past for installation in exhaust heat recovery systems [24,25,26]. Stobart and coworkers modeled and experimentally tested TE performance of devices with exhaust temperatures up to 800 K. Thermal asymmetry between the hot side and cold side accounting for the difference in heat transfer due to internal electric energy conversion was not considered by Stobart et.al. Their model was based on the average figure of merit (ZT) of the TE material that assumes optimal device geometry and optimal current [27,28]. Geometric parameters that affect TE performance include n/p-type leg area ratio, leg length, the area of individual legs and the distance between

adjacent legs. Modeling efforts by Hendricks et al. [16,19] considered temperature-dependent TE properties to determine optimal TE leg areas, lengths, and device designs. Modeling carried out by Xuan and Cheng [29,30,31], which dealt with thermoelectric coolers, considered the geometrical optimization of the length of TE legs only. Miller et al. studied heat transfer and heat exchanger optimization for a combined TE and organic Rankine cycle waste heat recovery system. Miller et al. calculated the TE device efficiency based on an average ZT for state of the art TE materials evaluated at typical operating temperatures [12]. A study by Matsubara et al. based on thermoelectric stacks composed of segmented legs projected highly efficient systems (up to 10%) [32]. Such efficiencies could produce enough power to supplement a vehicle alternator or replace it altogether [33,34]. Hussain et al. developed a model with thermally lumped TE devices that accounted for transient behavior and thermal asymmetry. In their model, the spatial variation and temperature dependence of the TE properties in individual TE devices was accounted for. Crane and coworkers developed a system level model integrating thermoelectric device and heat exchanger for cross-flow and counter-flow heat exchanger configurations. They used an analytical, thermally lumped TE leg performance model that correctly accounted for thermal asymmetry. Some of this modeling work was also validated with experimental results. Crane's most recent model incorporated transient performance. The report also presented a novel high-power-density subassembly of the TEG module that has various advantages over the conventional TEG assembly [18,35,36]. Kumar and coworkers presented a thermal resistance based numerical model to study the electrical power output and pressure drop for various flow rates for a GM prototype TE generator designed for a Chevrolet Suburban [37]. This study incorporated junction-averaged temperature dependent TE properties. A recent report by Espinoza and coworkers reported modeling efforts which take into account the temperature dependent properties along the heat exchanger, but not within the leg [38]. In addition, vehicle manufacturers including BMW, Ford, GM, and Ford have all been involved in studies on thermoelectric heat exchangers for automobiles in partnership with the U.S. Department of Energy [33].

Optimization studies on TEGs in the past have assumed a constant temperature boundary condition for simplified analysis. However, for a thermoelectric waste heat recovery system installed in the exhaust of a vehicle, these temperatures will be dictated by convection heat transfer on both sides (exhaust and coolant). Therefore, a more appropriate/realistic boundary condition to govern the performance of TEGs is convection heat transfer rather than constant temperature boundary conditions for TE junctions. Gomez et al. reported a modeling study incorporating constant reservoir temperatures instead of constant TE element junction temperatures. They discussed the relationship between fill fraction, leg length, and the ratio of load resistance and internal resistance for optimal performance [20]. However, their model did not account for temperature dependent TE properties. The study was carried out for low temperature ranges (~350K) based on experimentally measured TE properties of a commercial TEG module.

A recent study by Baker et al. introduced a finite difference system level numerical model for thermoelectric waste heat recovery system based on Mg and Mn silicides. The model correctly accounted for spatial- and temperature-variant properties. Heat exchanger model was supported by some experimental results as well. Pumping power required for the coolant flow was also taken into account. However, a validation of the device model was out of the scope of the study. In addition, the optimization study did not include the effect of thermal resistances. The device optimization results were provided for a given fill fraction and therefore, did not provide any trend for parameters under consideration [39,40,41].

None of the previous work, except the model introduced by Baker et al. discussed here used a TE model that accounted for spatial- and temperature-variant properties within the TE material of an individual TE couple. In addition, the majority of them have focused mainly on the optimization of the heat exchanger geometry only. The ones that discuss the optimization of TEG module geometry itself do not consider the interdependencies between the optimization variables like leg area ratio, area of

individual TE legs, spacing between the legs, and the load resistance. A comprehensive study regarding the optimization of TEG modules that includes such analysis is lacking.

The present study focuses primarily on modeling the performance and optimization of TEG modules integrated with a system level heat exchanger and using Mg_2Si and p-type $MnSi_{1.8}$ based silicides as the TE materials. A numerical model has been developed using a finite difference technique which accounts for spatial- and temperature- dependent thermoelectric properties. The model also couples hot side and cold side convection heat flux, thus accounting for the thermal asymmetry, using a numerical root finding algorithm. The model has been developed to understand and analyze the interdependencies among various parameters such as the height of the TEG modules, volume of TE material, the area ratio between n-type and p-type legs, and the load resistance. The set of parameters that provides the greatest power conversion for a given heat exchanger and operating conditions is determined. In addition, two TE leg pair devices, each with a different approach to fabrication, have been assembled, and an experimental setup was constructed to validate the numerical model. Electrical contact resistances was experimentally measured and incorporated into the numerical model for validation purposes. One unique feature of this study is that it is focused on Mg and Mn based silicides. These materials have potential cost advantages with comparable ZT values at higher temperatures [42], suitable for integration into heat exchangers employed in diesel and gasoline vehicles as opposed to conventional TE materials, e.g. bismuth telluride, lead telluride, and silicon germanium alloys.

The present study also presents a novel rearrangement of TE waste heat recovery systems. The negative impacts of thermal contact resistance on the performance of such waste heat recovery systems are well documented in literature [26]. The new system configuration model provides an approach to reduce the thermal contact resistances by half while increasing the overall efficiency of the system. A modeling comparative study between a conventional waste heat recovery system and the new system is discussed in detail.

Chapter 2: Modeling Approach

A 1-D finite difference TE device model and a heat exchanger model were developed to simulate their performance. Both models were coupled based on the total heat transfer between the heat exchanger and the TE device. The TE device was modeled at the TE leg pair level using an iterative finite difference scheme described by Hogan and Shih [43]. Reynolds and Nusselt number correlations for laminar and turbulent flow based on hydraulic diameters were used to model the heat exchangers of various geometrical configurations under study. Pressure drop was also incorporated into the heat exchanger to account for pumping requirements.

2.1 TEG Device Model

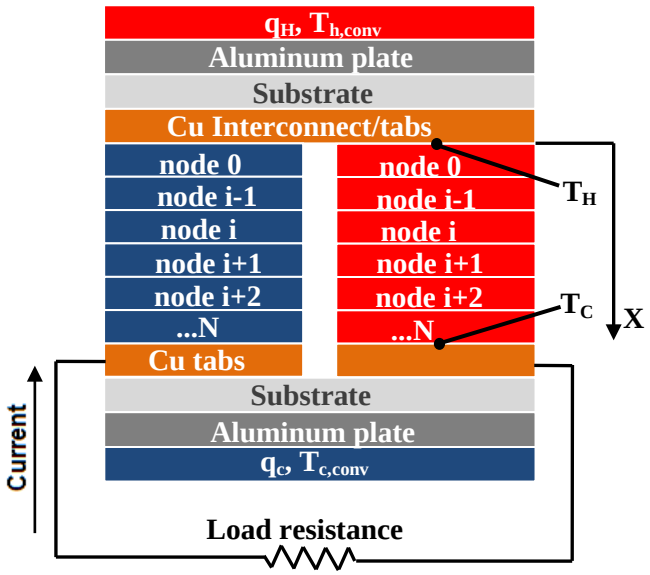


Fig. 2: Schematic of the numerical scheme for TE leg pair

To account for the temperature dependence of the TE properties in the direction of heat conduction and the thermal asymmetry, a TE device finite difference model was developed. A schematic of the finite difference numerical scheme is shown Fig. 2. The equations that govern the heat transfer and temperature distribution are:

$$\frac{\Delta T_i}{\Delta x} = \frac{1}{k_{i-1}}[JT_{i-1}\alpha_{i-1} - q_{i-1}] \quad (1)$$

$$\frac{\Delta q_i}{\Delta x} = \rho_{i-1}J^2[1 + Z_{i-1}T_{i-1}] - \frac{J\alpha_{i-1}q_{i-1}}{k_{i-1}} \quad (2)$$

In eq. (1) and (2), $\Delta T_i = T_i - T_{i-1}$, $\Delta q_i = q_i - q_{i-1}$, Δx is the distance between two adjacent segments, k_i is the temperature-dependent thermal conductivity evaluated in the middle of segment i, J is the electrical current density, $T_i$is the temperature in the middle of segment i, α_i is the temperature dependent Seebeck coefficient evaluated in the middle of segment i, q_i is the heat flux at the middle of segment i, and ρ_i is the temperature dependent electrical resistivity evaluated in the middle of segment i. Eq. (1) and (2) were solved for each segment of the leg. Seebeck voltage was calculated by integrating the Seebeck voltage at each node over the TE element as,

$$V_S = \sum_{i=1}^{n} \alpha(T_i - T_{i-1}) \quad (3)$$

Either load resistance or current density could be specified in the model. In this case, load resistance rather than current was input to the model since this is a parameter that needs to be specified for a given hardware configuration and can be easily controlled. Inputting load resistance rather than current, however, requires an additional set of iterations which is closed by the following equation [44]:

$$I = JA = \frac{\alpha\Delta T}{R_{load} + R_{internal}} \quad (4)$$

where A is the cross sectional area of either leg. The total power delivered to the load resistance is calculated by the expression:

$$P = I^2 R_{load} \quad (5)$$

For a pair of thermoelectric legs with convection heat transfer on both sides and isothermal interconnects with negligible electrical heating, the boundary conditions are combined heat flux and temperature. The interconnects, which are typically copper, are assumed to be isothermal because the thermal conductivity is high. Generation due to Joule heating is assumed to be negligible due to high electrical conductivity, and thermoelectric energy conversion at the interface of the interconnects and the TE

materials is neglected. On both the hot- and cold-side of the TE leg pair, the temperature of each leg must be equal to the temperature of the isothermal interconnect interface, and this is also the temperature driving conduction heat flux. The total composite heat flow is given by:

$$Q_{comp} = A_n q_n + A_p q_p \tag{6}$$

where A is the cross-sectional area of the n- or p-type leg, Q_{comp} is the total composite heat flow, and q is the heat flux in the specified leg. Since the governing equations are given in terms of heat flux, the TE device level is modeled using this variable at the leg pair level. However, this heat flux is converted to total heat transfer when the TE device model and the heat exchanger model are coupled together. A visual representation of the p-type, n-type and void areas is shown in Fig. 3. The void area is the empty space between adjacent TE legs, and it is assumed to be perfectly insulated.

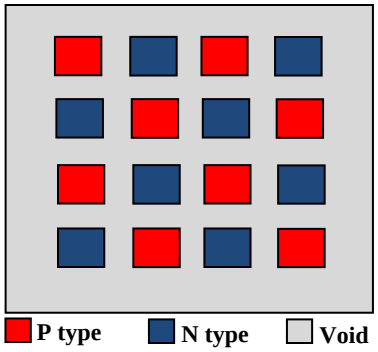


Fig. 3: Schematic of a TE leg layout areas with void space

The total composite heat flow must be equal to the total heat flow to or from the hot and cold heat exchangers to/from either the hot or cold side of the TE element, given by,

$$Q_{h,comp} = \frac{\left(T_{h,conv} - T_h\right)}{R_h} \tag{7}$$

for the hot side, and

$$Q_{c,comp} = \frac{\left(T_c - T_{c,conv}\right)}{R_c} \tag{8}$$

for the cold side, where R is the overall heat transfer resistance for the hot or cold side (including thermal resistance due to conduction and contact resistances associated with the aluminum plate, substrate, and interconnect as well as thermal resistance due to convection as explained in the heat exchanger model section), T_{conv} is the temperature of the hot or cold fluid, T is the temperature of the hot or cold side of the TE leg pair, and Q_{comp} is the total composite heat flow on the hot or cold side of the TE device, given in equation. (5). Subscripts h and c indicate whether the variable corresponds to the hot or cold side.

The temperature boundary conditions are specified as follows:

$$T_{n,h} = T_{p,h} = T_h \tag{9}$$

$$T_{n,c} = T_{p,c} = T_c \tag{10}$$

where, as before, subscripts n and p indicate n- or p-type leg, and subscripts c and h indicate cold or hot side. The parameter used in this study to characterize the coverage of TE materials present in a given area of the module, is termed the fill fraction, and is given by:

$$FF = fill\ fraction = \frac{A_n + A_p}{A_n + A_p + A_{void}} \tag{11}$$

For all the numerical modeling schemes, an open source programming package Python was used on a Linux platform. In-built Python functions were used for various purposes. Function fsolve() was used for iterative purposes. This function minimized the error while iteratively solving the discretized finite difference equations (1) and (2). Function fmin() was used for optimization study. Python function fmin() minimizes a function value by varying the parameters. In this case, negative of power per unit area was the function value which was minimized by varying four parameters namely length, fill fraction, leg area ratio, and current.

2.2 Heat Exchanger Model

A 1-D finite difference heat exchanger model was developed to complete the system level configuration of the TE devices. Well established correlations found in the literature for Reynolds number, Re_D, and Nusselt number, Nu_D for laminar and turbulent flow were used for this purpose. Calculations were done based on hydraulic diameter of the ducts. The correlations are given below [45].

$$Nu_D = \frac{8.24}{4.36} 0.023 Re_D^{4/5} Pr^{1/3} \tag{12}$$

for turbulent flow ($Re_D > 2300$) with heat transfer on both sides.

$$Nu_D = \frac{5.39}{4.36} 0.023 Re_D^{4/5} Pr^{1/3} \tag{13}$$

for turbulent flow ($Re_D > 2300$) with heat transfer on one side only.

$$Nu_D = 7.54 \tag{14}$$

for laminar flow ($Re_D < 2300$) with heat transfer on both sides.

$$Nu_D = 5.39 \tag{15}$$

for laminar flow ($Re_D < 2300$) with heat transfer on one side only. Here, Pr is the Prandtl number. Friction factors, f, were calculated using the following set of equations.

$$f = \frac{24}{16} 0.078 Re_D^{-1/4} \tag{16}$$

for turbulent flow ($Re_D > 2300$), and

$$f = \frac{24}{Re_D} \tag{17}$$

for laminar flow ($Re_D < 2300$).

Where, Re_D is the Reynolds number based on the hydraulic diameter, and the hydraulic diameter, D_h, was calculated based on the specific duct and fin geometry as,

$$D_h = \frac{4A_C}{P} \tag{18}$$

Where, A_C is the cross sectional area of the duct and P is the wetted perimeter. The leading coefficients in equations (12), (13), and (14) account for the rectangular shape of

the ducts, since the original equations were derived for round tubes. These scaling factors were found in Table 3.2 of Bejan's Convection Heat Transfer [45]: A schematic of a rectangular duct heat exchanger with straight fins is given in Fig 4.

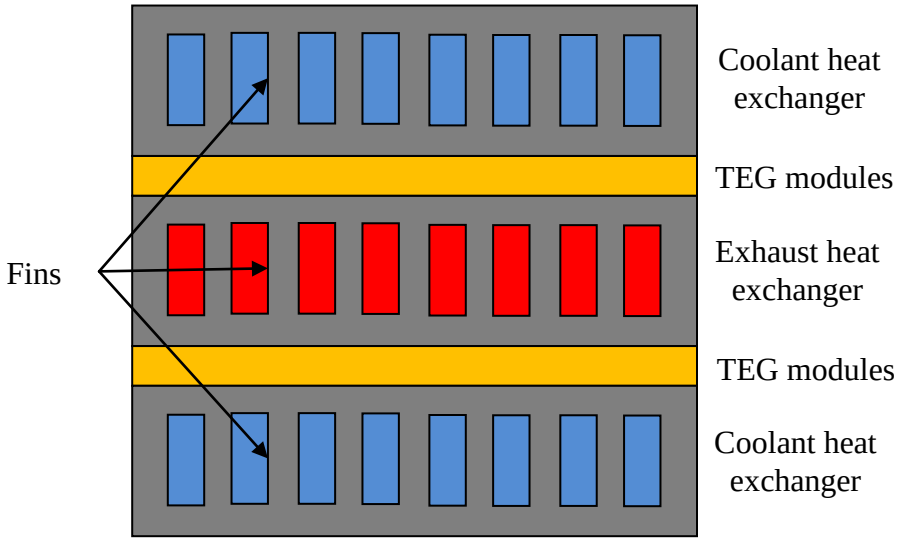


Fig 4: Schematic of a straight fin rectangular duct heat exchanger. Coolant flows through the two ducts at the top and the bottom (blue). Hot fluid passes through the duct in the center (red).

The heat exchanger was divided into a finite number of equally spaced nodes in the streamwise direction. The mesh density (number of nodes) was chosen to ensure grid independence of the solution. At each node, the temperature of the exhaust and coolant were used as the boundary conditions for the TE device model. Each node in the streamwise direction of heat exchanger stored the values of exhaust and coolant temperatures as well as the hot-side to cold-side temperature distributions within the TE elements. The flow properties used in the model were calculated as a function of temperature as described in Appendix A. Temperature dependent TE properties were measured and evaluated at every temperature in the model using an appropriate curve fit.

The convection heat transfer coefficient, h, was calculated using the equation,

$$h = \frac{Nu_D k}{D} \tag{19}$$

Where, k is the thermal conductivity of the fluid. Pumping power required to overcome the pressure drop was also incorporated into the model and subtracted from the raw power output of the TE devices to give the net system power output. Pressure drop for a mean fluid velocity, U, was calculated using,

$$\frac{\Delta P}{\Delta x} = f \frac{P}{A_C}\left(\frac{1}{2}\rho U^2\right) \tag{20}$$

Pumping power at each node, $\dot{W}$, was calculated using the following equation:

$$\dot{W} = \dot{V}\dot{\Delta P_l} \tag{21}$$

where, $\dot{V}$ is the volume flow rate at the given node and $\dot{\Delta P_l}$ is the pressure drop at the given node. The sum of all the nodal pressure drops was taken as the total pumping power required for the system.

For the finite difference discretization, a first order Euler scheme was utilized in the streamwise direction of the heat exchanger. The equations used to calculate the temperature at each new node in the streamwise direction, the following equation was used:

$$T_{exh,i} = T_{exh,i-1} + \frac{Q_{exh,i-1}}{C_{exh}} \tag{22}$$

$$T_{cool,i} = T_{cool,i-1} + \frac{Q_{cool,i-1}}{C_{cool}} \tag{23}$$

where, T_i is the temperature is the present node, T_{i-1} is the temperature in the previous node, Q_{i-1} is the heat transfer in the previous node and C is the heat capacity. Subscripts exh and $cool$ refer to exhaust and coolant respectively.

2.3 System Level Thermal Circuit

Numerous thermal resistances are present between the exhaust gas or the coolant and the hot or cold side of the TE elements, as shown in the thermal circuit diagram in Fig. 5. These thermal resistances can be lumped into one single thermal resistance. Since the contact area of various elements inside the TE devices changes as fill fraction is

changed, the resistances have to be normalized based on their respective areas before they can be lumped together.

$$R_{thermal} = R_{convection} + R_{Al\ plate} + R_{substrate} + R_{interconnect} + R_{contacts} \quad (24)$$

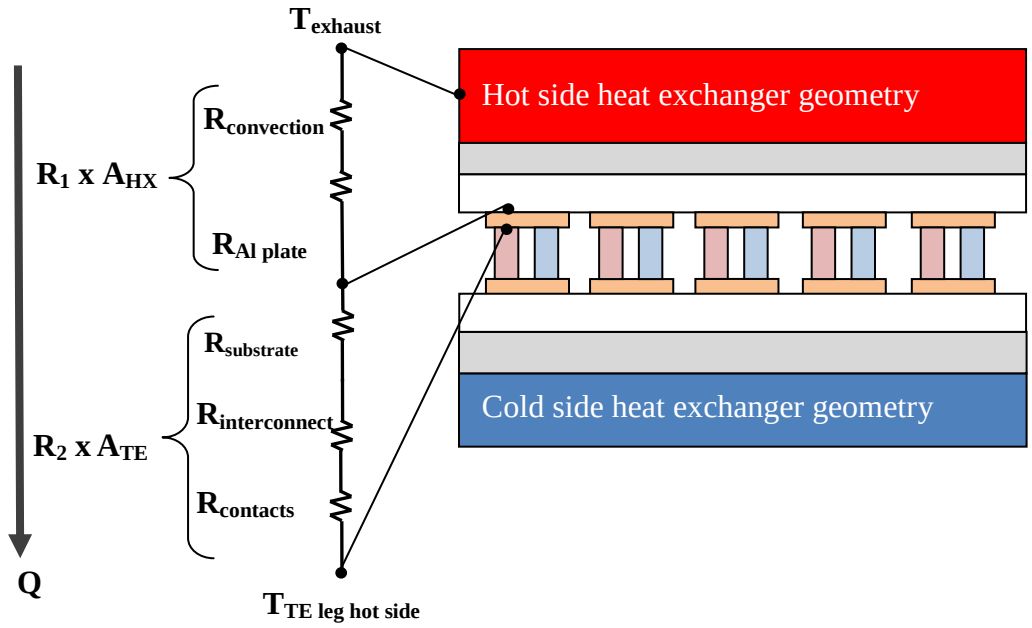


Fig. 5: System level integration of Thermoelectric modules into the heat exchanger. Thermal resistances present between the exhaust gas and the hot side of the TE element are shown here. Similar thermal circuit exists on the cold side as well.

The individual resistances were calculated using the following equations.

$$R_{convection} = \frac{1}{hA_{HX}} \quad (25)$$

$$R_{Al\ plate} = \frac{t_{Al\ plate}}{k_{Al\ plate}A_{HX}} \quad (26)$$

$$R_{substrate} = \frac{t_{substrate}}{k_{substrate}A_{HX}} \quad (27)$$

$$R_{interconnect} = \frac{t_{Cu\ interconnect}}{k_{Cu\ interconnect}A_{TE\ pair}} \quad (28)$$

$$R_{contact} = R''_{contact}A_{TE\ pair} \quad (29)$$

Above mentioned thermal resistances were calculated for a typical thickness of heat exchanger aluminum plate, ceramic substrate, and copper interconnect. Contact resistance has been found to be a strong function of operating temperature, clamping pressure, surface finish, and material composition, among other variables [46,47,48]. After a comprehensive literature review, a combined contact resistance in the range of 0.0001 m^2-K/W and 0.00001 m^2-K/W was considered reasonable for modeling purposes in this study (R_2 = 0.0001 - 0.00001 m^2-K/W in Fig. 5). Contact resistances for three different interfaces (TE element/Cu interconnect, Cu interconnect/substrate, substrate/Al plate) were lumped into one value. Resistance due to convection and heat transfer enhancement due to finned geometry was calculated as explained in the next section. Thermal resistances of various components constituting the thermal circuit is provided in Table 1.

Table 1: Thermal resistance values considered for the modeling study.

Type	Thermal resistance (m^2-K/kW)
Interconnect	0.00075 (k = 400 W/m-K)
Substrate	0.0005 (k = 200 W/m-K)
Al plate	0.0317 (k = 200 W/m-K)
Convection	Based on HX geometry
Contact	0.0001 - 0.00001

2.4 Heat Exchanger Enhancement

Heat exchanger fin configurations in this study were modeled using finned surface theory [49]. A schematic of a rectangular fin is shown in Fig 6. The heat transfer coefficient was first calculated for the geometrical configuration based on hydraulic diameter (equation (18)). Fin efficiency and area of the fins were then used to calculate the composite heat transfer coefficient. This heat transfer coefficient provides the thermal resistance due to convection, which is an important parameter in the system level thermal circuit shown in Fig 5.

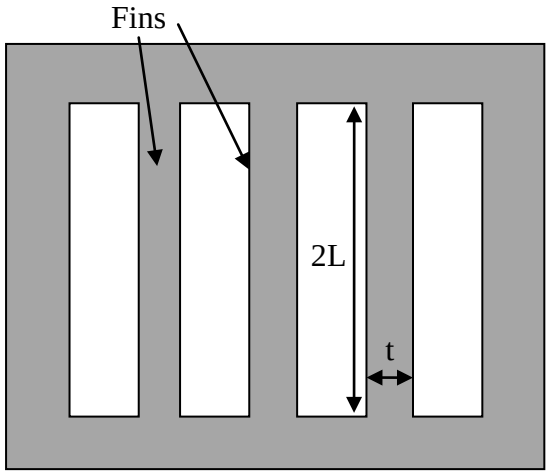


Fig 6: Schematic of a straight fin arrangement in a rectangular duct.

Fin efficiency can be calculated using the formula,

$$\eta_{fin} = \frac{\tan\theta}{\theta} \tag{30}$$

where,

$$\theta = mL \tag{31}$$

and,

$$m = \sqrt{2h/(k_{fin}t)} \tag{32}$$

where, L is the length of the fin, h is the convection coefficient based on the hydraulic diameter, k_{fin} is the thermal conductivity of the fin material, t is the individual fin thickness.

The composite heat transfer coefficient, h_{comp}, can be subsequently calculated using the equation,

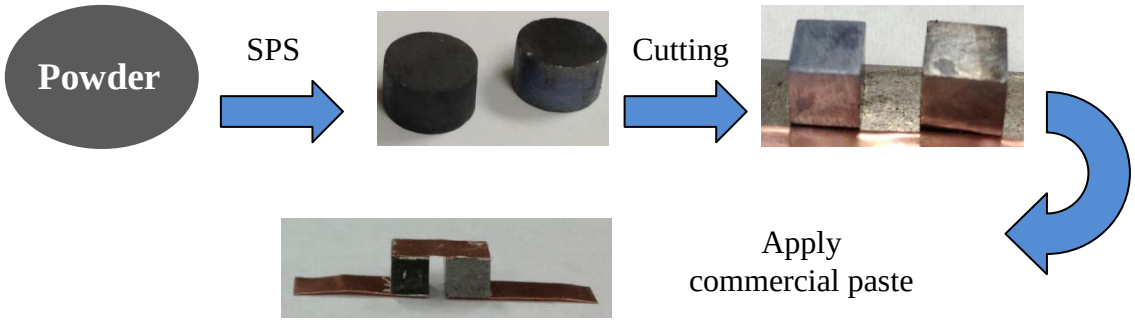
$$h_{comp} = \frac{\eta_{fin}hA_{finned} + hA_{unfinned}}{A_{finned} + A_{unfinned}} \tag{33}$$

where, A_{finned} is the total surface area of the finned surfaces, and $A_{unfinned}$ is the total surface area of the unfinned surfaces.

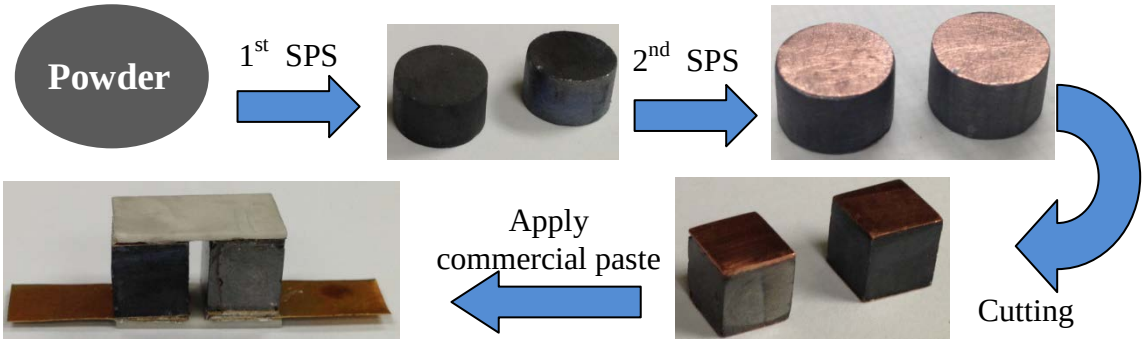
Chapter 3: 1st and 2nd Generation TE Device Experiments

3.1 Device Fabrication

Two different TE devices - the 1st and 2nd generation, each composed of a single TE leg pair were fabricated for the purpose of experimental model validation and electrical contact resistance measurements. The compositions of the TE elements used in the 1st and 2nd generation TE devices are shown in Table 2. The dimensions of the TE elements (also referred to as TE legs) are listed in Table 3. For both the 1st and 2nd generation TE devices, an appropriate composition of each element in powder form was mixed together homogenously. Pellets were made from this homogenous composite powder using a press. The pellets were then heated in a furnace to allow solid state



(a) 1st generation TE device



(b) 2nd generation TE device

Fig. 7: Fabrication procedure for the 1st and the 2nd generation TE devices outlined.

reactions to take place. This solid block was again ground to powder form which was used in the SPS (Spark Plasma Sintering) machine to fabricate the final TE element samples. The temperature used for SPS process for the p-type and the n-type materials was around 1000°C and 850°C, respectively. Since the die used was cylindrical in shape, the n-type and p-type samples were cut into square blocks using a diamond saw so that a rectangular TE device could be constructed.

Table 2: TE Materials used in 1st and 2nd generation TE device fabrication

Material	Type	Composition
HMS	p-type	$Mn(Al_{0.0015}Si_{0.9985})_{1.8}$
MgSi	n-type	$Mg_2(Si_{0.4}Sn_{0.4}Ge_{0.2})_{0.985}Sb_{0.015}$

Table 3: Dimensions of TE legs used in 1st and 2nd generation TE devices

Dimension	1st generation		2nd generation	
	p-type	n-type	p-type	n-type
Base area (mm^2)	5.22 x 5.42	4.52 x 5.26	5.48 x 5.35	5.56 x 5.65
Height (mm)	4.97	4.97	6.22	6.22

The 1st generation TE device was constructed using a one step SPS process. The solid TE element samples were first made from powder using SPS. The 1st generation TE device was then assembled by bonding copper interconnects directly to the TE elements using a commercial silver paste with high thermal and electrical conductivity. Copper interconnects were used because of their low electrical and thermal resistance. The bonded piece was sandwiched between electrically insulating and thermally conducting ceramic substrates. It was ensured that the TE elements were connected in series electrically and in parallel thermally.

An improved 2nd generation TE device was constructed using a two-step SPS process. First, the p-type and the n-type solid TE element samples were prepared using SPS. In the second step, thin copper sheets (~ 1 mm thick) were cut into circular shapes having a diameter the same as the die and SPS was used again to bond the copper electrodes to the TE elements. The sintering temperatures for the n-type and the p-type samples were 650°C and 600°C, respectively. Samples were then cut into rectangular

blocks, and commercial silver paste was used to assemble the whole device. Unlike in the 1st generation TE device, where the commercial paste was used for Cu-TE element bonding, the paste was used in this case solely for Cu-Cu bonding. As mentioned above, the Cu-TE element bonding was made using a second sintering process in the 2nd generation TE device. The device was then sandwiched between ceramic substrates. The fabrication procedures are outlined in Fig. 7 and photos of both the devices are shown in Fig. 8.

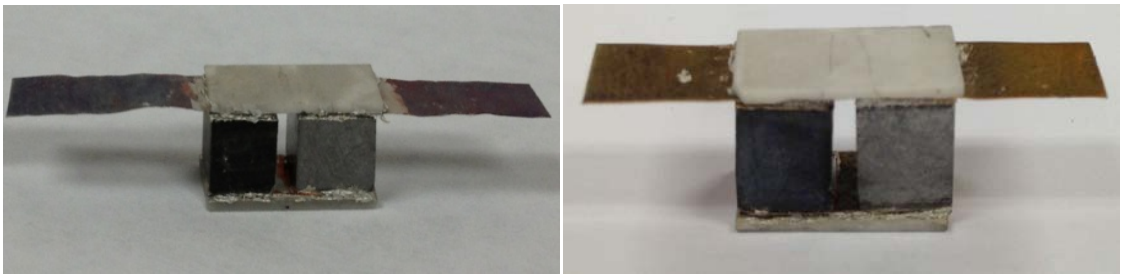


Fig. 8: The 1st generation TE device (left)was fabricated using commercial silver paste for electrode bonding while SPS was used for electrode bonding for the 2nd generation TE device (right).

3.2 Experimental Methods

3.2.1 Electrical Resistance Measurement

The four probe measurement technique was used to measure the electrical resistance of TE elements as well as the contact resistances for the 1st and the 2nd generation TE devices. Four probe technique is a process of measuring electrical resistance more accurately compared to traditional methods by isolating the current carrying probes and voltage measuring probes. Current is passed through the test sample using two current probes while the voltage drop across the specimen is measured using voltage probes. The slope of the Current-Voltage (I-V) plot gives the electrical resistance.

Thus measured contact resistance values were incorporated into the model for the purpose of model validation.

For the 1st generation TE device, the electrical resistance of the elements was calculated using experimentally measured properties. The contact resistance of the device was then found by subtracting the calculated TE element resistance from the total device resistance that was experimentally measured. This was possible because the contact resistances were dominant in the overall measurement. The total device resistance was measured by passing various magnitudes of positive and negative DC current through the device, and measuring the corresponding voltage drops. The total device resistance was given by the slope of the I-V curve.

For the 2nd generation TE device, the internal resistance as well as the contact resistance was experimentally measured. Using a low electrical and thermal resistance paste, copper electrodes were bonded to each of the leg interconnects. Various magnitudes of positive and negative DC current was passed through the devices and voltage drops at various locations on the device were measured. The slope of the I-V curve was taken as the resistance. Temperatures of the hot and cold sides were recorded to account for Peltier heating. However, the temperature differences were negligible for the small current flows used.

3.2.2 Power Output Measurement and Model Validation

An experimental setup was designed and constructed to validate the numerical TEG device model. The setup is shown in Fig. 9. A one-dimensional heat flow device was made using a hot plate as the heat source and a circulating water heat exchanger as the heat sink. A circulating water heat exchanger was designed and fabricated from aluminum. The TE device was placed in between the heating and the cooling sections. Thermally conducting paste was used at the interface between the TE device and the heat source and sink, while the heating and cooling sides were both thermally insulated to facilitate larger temperature drop across the TE devices.

Table 4: List of all the equipments used in the experiments

Equipment	Type
Heater	Kitchen Gourmet 1000 W
Thermocouple monitor	SRS Model SR630
Digital multimeter	Agilent 34401A
Current source	Keithley 6221
Variac	Staco energy 1.4 kVA
Thermally conductive paste	Omegatherm 201
Thermally and electrically conductive paste	Aremco Pyroduct 597-A

K-type and T-type thermocouple wires were bonded to the surface of both n-type and p-type TE legs near the hot and cold edges for the 1st generation and the 2nd generation TE devices, respectively. Heating load was varied using a variac transformer and the corresponding hot and cold side temperatures of TE legs were measured. Open circuit voltage and the voltage drop across a known load resistance were also measured using a high accuracy voltage meter. The load resistance was a thin constantan wire. The values of the electrical resistances were 6.728 and 2.220 ohms for the 1st and the 2nd generation devices, respectively. All the equipments used in the experimental setup are listed in Table 4. The power output of the device was calculated using the following equation:

$$P = I^2(R_{load} + R_{contact}) \tag{34}$$

Where, I is the current flowing through the circuit, which was calculated experimentally by measuring the voltage drop across the known load resistance. This is the total power output of the device delivered to both the load resistance and the contact resistance. Water flow rate was varied and set for the maximum cooling effect. Every time the heating load was varied, the device was left for about 45 minutes to reach steady state condition. The numerical model was modified for the specified temperature boundary conditions for comparison with the experiments.

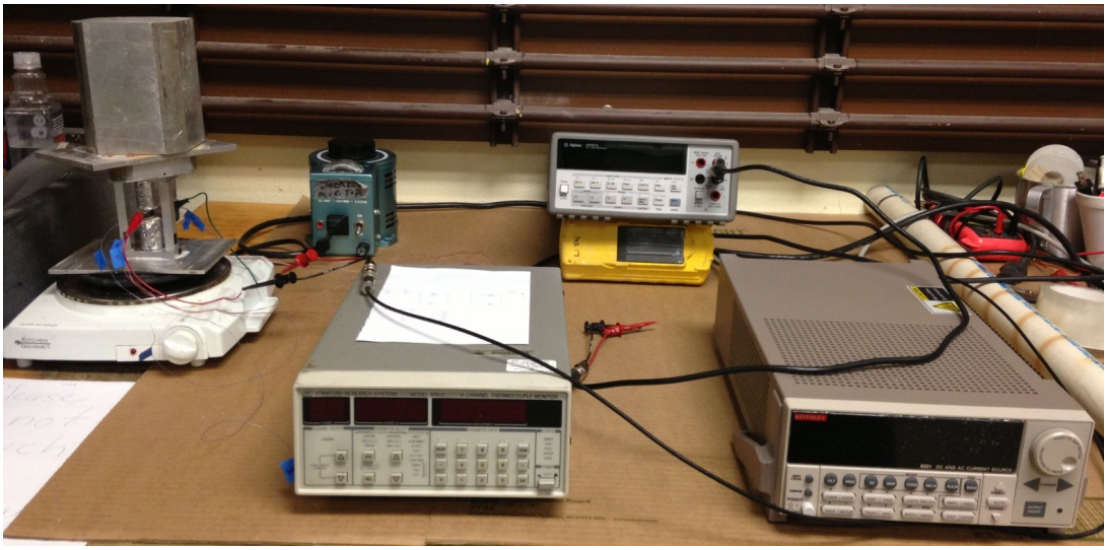


Fig. 9: Experimental setup for 1st and 2nd generation TE device power output measurement.

3.3 Experimental Results

3.3.1 Electrical Resistance

Fig. 10 shows the experimentally measured values of electrical resistance of the TE elements as well as the electrical contact resistance between the elements of the devices. The I-V plot used to deduce electrical contact resistance for the 2nd generation TE device is shown in Fig. 11. The results, as shown in Fig. 10, show a very large decrease of almost two orders of magnitude in contact resistance when the SPS technique was used for bonding the copper electrodes to TE elements versus bonding with the paste. As the sintering temperature was increased, better bonding between the TE elements and the Cu electrode was observed. However, above some high enough temperature, the Cu electrodes melted and formed silicides. The optimum sintering conditions for such bonding processes may be expected to vary for different TE materials. Finding the optimum SPS conditions can have an impact on the quality of

bonding. Contact resistance can be decreased further by finding the optimum sintering conditions for electrode-TE element bonding.

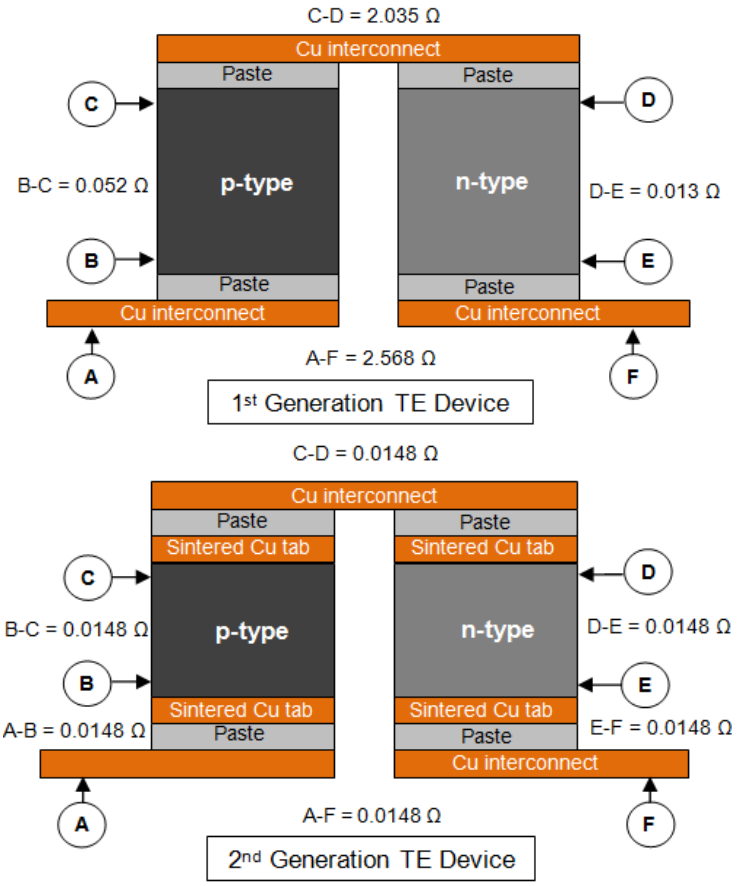


Fig. 10: Experimentally measured electrical resistance of constituting elements of the 1st generation TE device (left) and the 2nd generation TE device (right) in ohms. Total electrical resistance of the 1st device is 2.568 ohms and that of the 2nd is 0.0148 ohms.

The four probe measurement technique is applicable and most suited to 1-D structures. The experimental measurements done in this study are on cubic blocks of silicide TE materials. The voltage reading at the surface location where the copper electrodes are bonded may have a different value than the average voltage across the cross section at that location. This may violate the 1-D assumption. Therefore, the uncertainties in the measured electrical resistances of the TE elements themselves may have an additional uncertainty due to these possible 3-D effects. However, since the

electrical resistance of the TE elements in the first device is negligible compared to the total resistance of the circuit, it does not does lead to significant errors in the power output analysis. Similarly, in the 2nd generation TE device, the electrical resistance of the TE elements and the contact resistance, which are measured experimentally, are negligible compared to the load resistance. So they do not lead to significant errors in the power output analysis. Data pertaining to error analysis in electrical resistance measurement is provided in Appendix B.

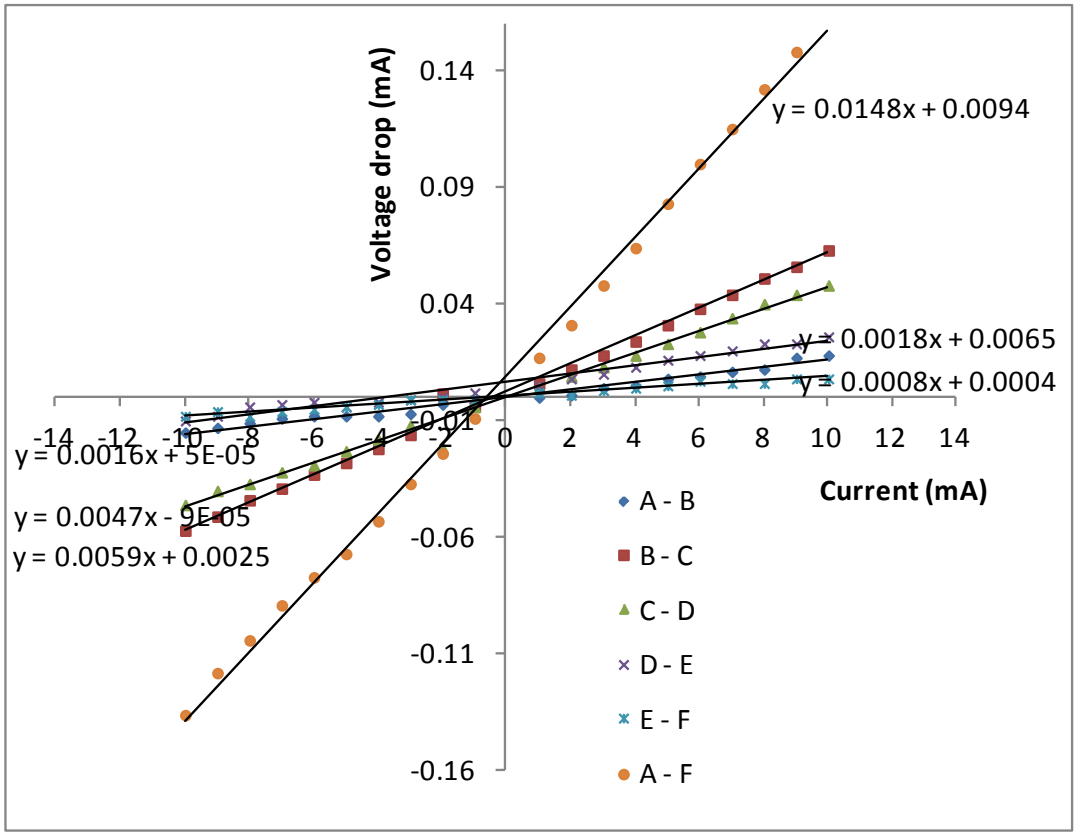


Fig. 11: Voltage drops between different locations on the 2nd generation TE device for various values of constant current flow. The slope of linear fit was used to deduce resistance values. Refer to Fig. 10 for probe locations A through F.

3.3.2 Model Validation

Fig. 12 shows the numerical and experimental electrical power output for both the 1st and 2nd generation TE devices as a function of the temperature difference between the hot and the cold sides. Power delivered by the TE device to the combined electrical load resistance and electrical contact resistance is plotted. Electrical contact resistance was incorporated as an additional resistance in the electrical circuit. Thermal contact resistance was not incorporated for this purpose because the temperatures were recorded directly on the TE elements. Because of the significantly lower electrical contact resistance of the 2nd generation device, its power delivery was significantly greater than that of the 1st generation device. The numerical model under predicts the power output for the 1st and 2nd generation TE device by an average of 20% and 23%, respectively, over the range of temperatures measured. Because of heat loss from the surfaces of the TE elements (where the thermocouples were attached), the average hot and cold side temperatures measured are expected to be lower than the actual area average temperatures, such that the temperatures in the center were higher than the temperatures at the surface (an attempt was made to quantify this error, which is discussed later in this section). In addition, because of geometrical constraints, the thermocouples were bonded to the sides of each TE element, a small distance away from the top and bottom of the TE element. Therefore, the temperature difference measured by thermocouples was slightly lower than the actual temperature difference across the entire TE element. Since the model calculates the Seebeck voltage and the corresponding power output based on the measured temperature difference, a lower temperature difference is expected to result in a lower Seebeck voltage and therefore lower power output.

An attempt was made to measure the amount of lateral heat loss. This was done by measuring the temperature drop in the radial direction on a borosilicate (glass) sample, whose dimensions were the same as that of the TE elements (shown in Fig. 13). Borosilicate was chosen because drilling holes into the TE elements was not a viable option. Thermal conductivity of borosilicate is nearly equal to the thermal conductivity of

TE elements under consideration. The thermal conductivity of each TE element is a strong function of temperature, however, it is approximately 2 W/m-K at room temperatures. The thermal conductivity of borosilicate is 1.14 W/m-K. Thermocouples were inserted to the center of the sample through a drilled hole whereas some were attached to the outside surface. Using the same test rig as described above, the

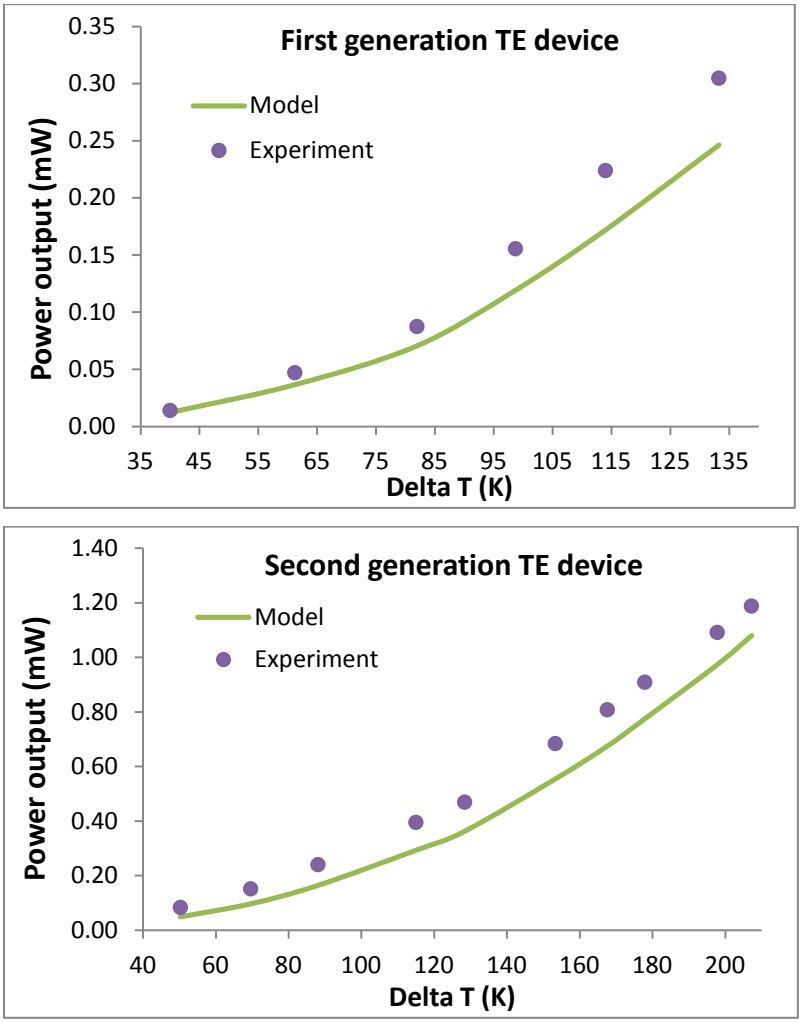


Fig. 12: Modeling and experimental results for power delivered to the electrical load for various temperature differences between the hot and cold sides of the TE elements for the 1st generation and the 2nd generation TE devices. The cold side temperatures were in the range of 315 K - 421 K while the hot side temperatures varied in the range of 355 K - 629 K

Fig. 13:Borosilicate sample for measuring the temperature drop in the lateral direction for a given temperature drop in the axial direction (two holes were drilled up to the center of the glass sample for thermocouple insertion).

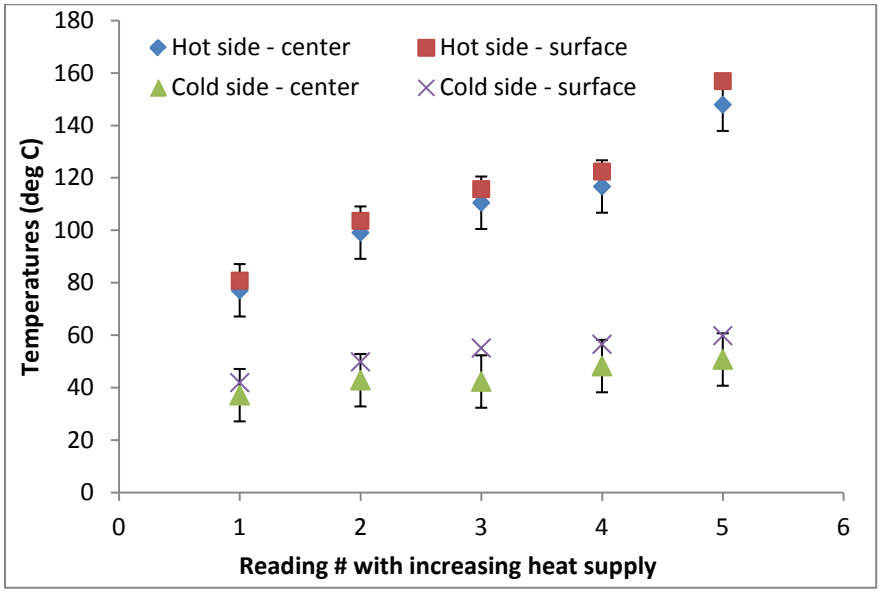


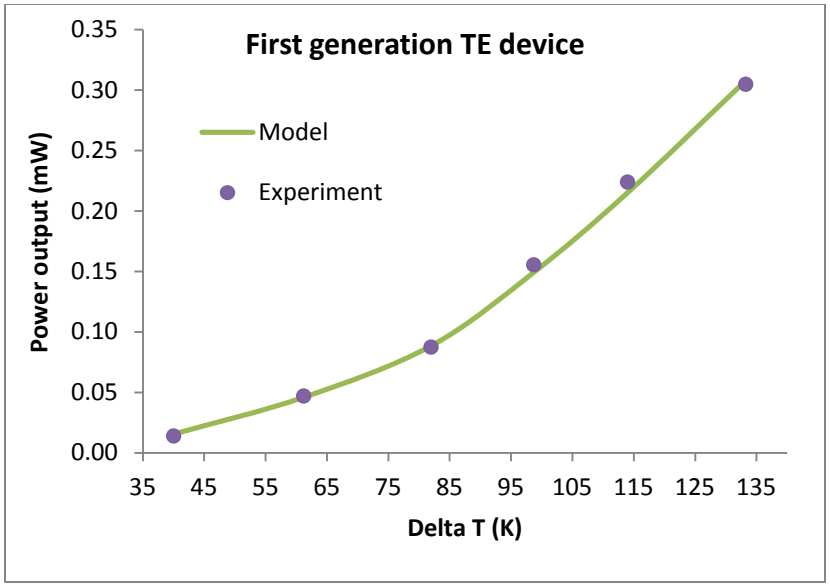
Fig. 14:Temperature readings for the hot and the cold side of borosilicate sample at the surface and the center. The readings were taken for five different heating loads.

temperature drop in the axial as well as the lateral direction were recorded. The results showed that the temperature difference in the lateral (radial) direction was within the uncertainty of the temperature measurements and small compared to the axial

temperature drop. As shown in Fig. 14, the temperature at the surface and the center of the sample is the same within the uncertainty on both the hot and the cold sides.

This conclusion suggests that the error in the power output has to come from other sources mentioned above. One of them is the TE property (Seebeck) deterioration due to sintering process. Using SPS to sinter copper electrodes directly to the TE elements at high temperatures can lead to diffusion of copper into the sample. This diffusion contaminates the sample and may deteriorate the thermoelectric properties reducing the Seebeck coefficient. It was observed that the Seebeck coefficient of TE element samples after sintering process were approximately only 70% of the original Seebeck coefficient values measured on pure samples. In addition, variation of TE properties of bulk Mg and Mn based materials from sample to sample, as well as oxidation of the outer surfaces under room conditions can adversely impact device power output.

The other source of error is the measurement of temperature slightly away from the edges of the TE elements. Hot and cold side temperatures of TE elements measured using thermocouples were extrapolated linearly to the outer edges of the TE elements to address this issue and to obtain a better estimate of the temperature difference the TE elements were subjected to. The power output comparison based on this extrapolation is shown in Fig 15. This extrapolation results in a significant improvement for the 1^{st} generation experiments. The model follows the experimental power output to within 3.8% over the range of temperatures. The same cannot be said about the 2^{nd} generation device for which the model over-predicts the power output increasingly as the temperature difference is increased. This is because, unlike in the 1^{st} generation TE device, the Seebeck coefficient of the TE elements was significantly reduced due to the sintering process. Therefore, the experimental power is lower than the numerical prediction, even though the temperatures were extrapolated.



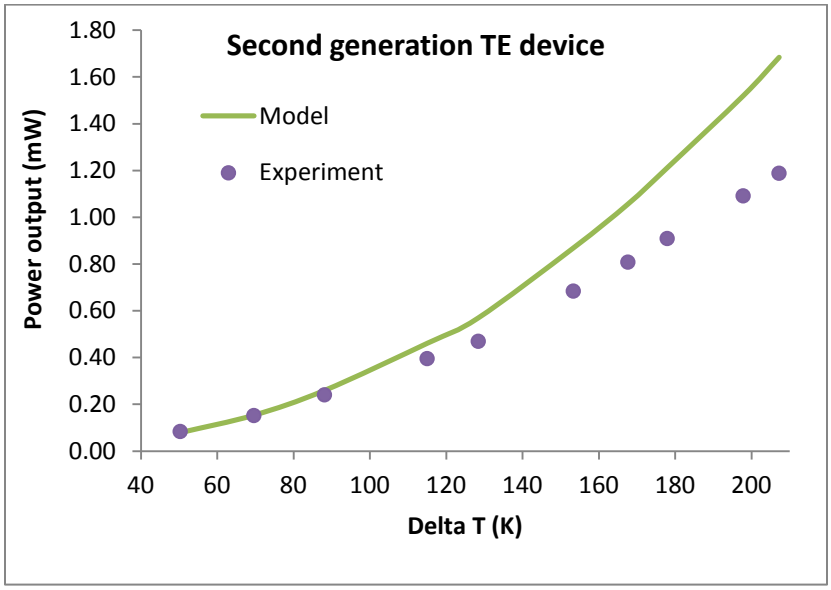


Fig. 15: Modeling and experimental results for power delivered to the electrical load for various temperature differences between the hot and cold sides of the TE elements for the 1st generation and the 2nd generation TE devices. The hot side and the cold side temperatures were linearly extrapolated to account for geometrical constraints on thermocouple placement on the TE samples.

Chapter 4: New System Design

Conventional waste heat recovery thermoelectric heat exchangers typically consist of a hot side heat exchanger and a cold side heat exchanger, with TEG modules sandwiched in between [16,17,20,35,36]. Since the power output is directly proportional to the temperature difference between the hot side and the cold side of the TE elements, presence of any unwanted thermal resistance between the TE elements and the heat exchangers can adversely affect (lower) the temperature gradient in the TE elements, and hence, the performance. Therefore, it is desired to increase the thermal resistance of the TE elements compared to any other thermal resistances that are present between the coolant and the exhaust. One of the major issues in the system level design of TE heat exchangers is dealing with thermal contact resistance. Thermal contact resistance at the interface of TEG modules and the hot and cold side heat exchangers can hinder the performance of TE waste heat recovery system. Reducing this contact resistance is an avenue which can lead to increased performance of TE heat exchangers.

The present study proposes a heat exchanger design where the contact resistance can be reduced by half simply by a novel arrangement of the TEG modules. A comparative study between conventional and the proposed new heat exchanger has been conducted. The study was independent of the TEG module optimization and was based on typical design parameters for the TEG devices used. A conventional system level TE heat exchanger arrangement is shown in Fig. 16. The heat exchanger design proposed in this study is shown in Fig. 17. Both the conventional and new heat exchanger designs have a finned heat exchanger geometry for the coolant and the exhaust side. The total volume occupied by both the heat exchangers was taken to be approximately 29.5 L, which was the size of the heat exchanger used in the previous experimental work [39]. Results suggest that a TEG package of this size can produce significant amount of electrical energy from vehicle waste heat. An optimization study on the size of the TEG package is out of the scope of this study.

The new heat exchanger requires TE modules having a modified geometry, as well. One of the two surfaces of the modules must have a finned geometry as shown in the zoom-in view in Fig. 17. For this purpose, the electrically insulating and thermally conducting ceramic substrate on one of the sides needs to be fabricated in such a way that the surface has protrusions in the form

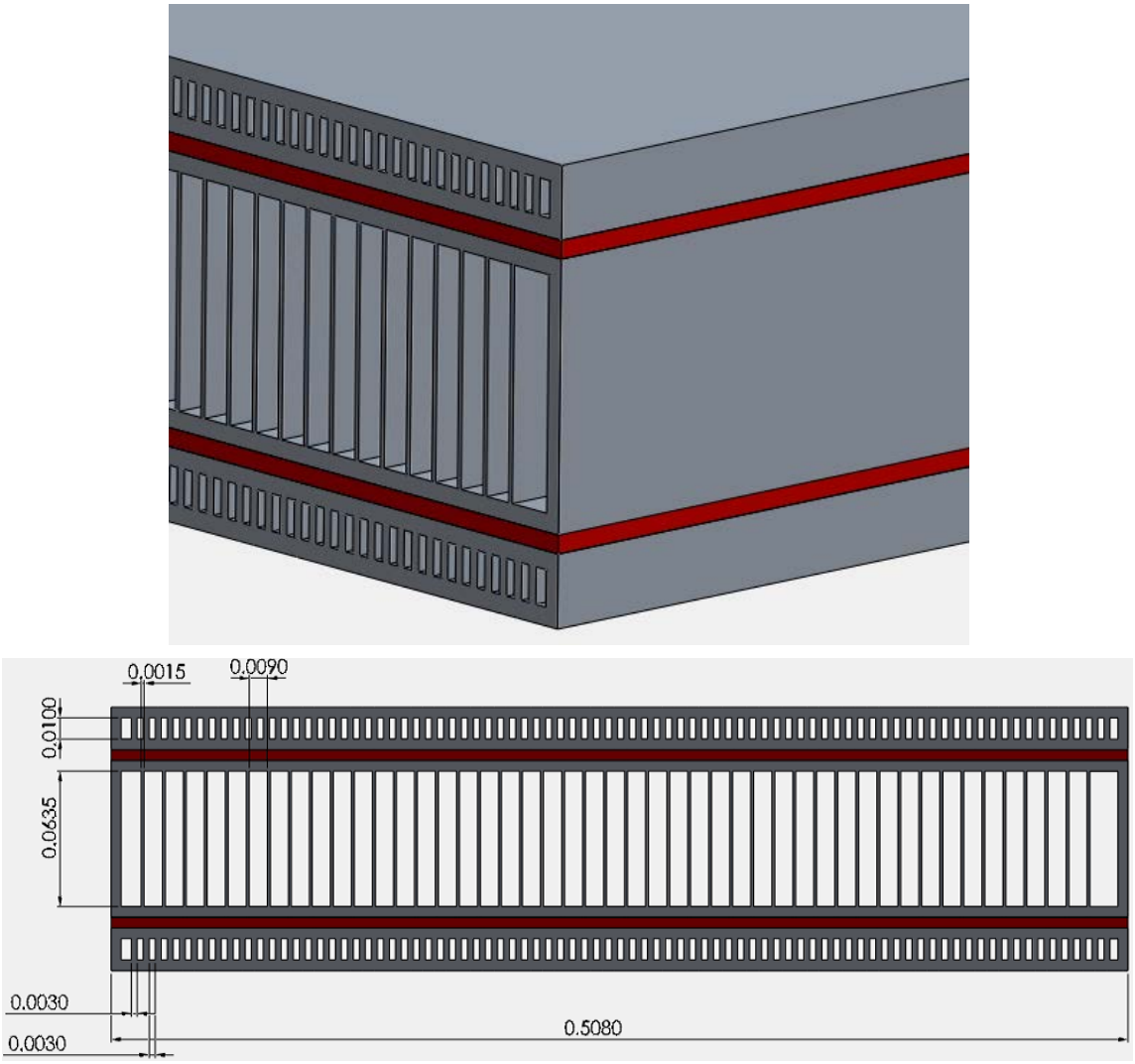


Fig 16: Conventional thermoelectric waste heat recovery system for automotive applications. TEG modules (red) are sandwiched in between coolant and the exhaust side (all dimensions are in meters. Heat exchanger length is 0.508 m).

of a fin. In addition, the ceramic substrates used here must have high thermal conductivity. Recent advancements in ceramic technologies have made it possible to manufacture highly conductive substrates for semiconductor industry. Substrate materials having thermal conductivities nearly as high as Aluminum are available commercially [50,51,52].

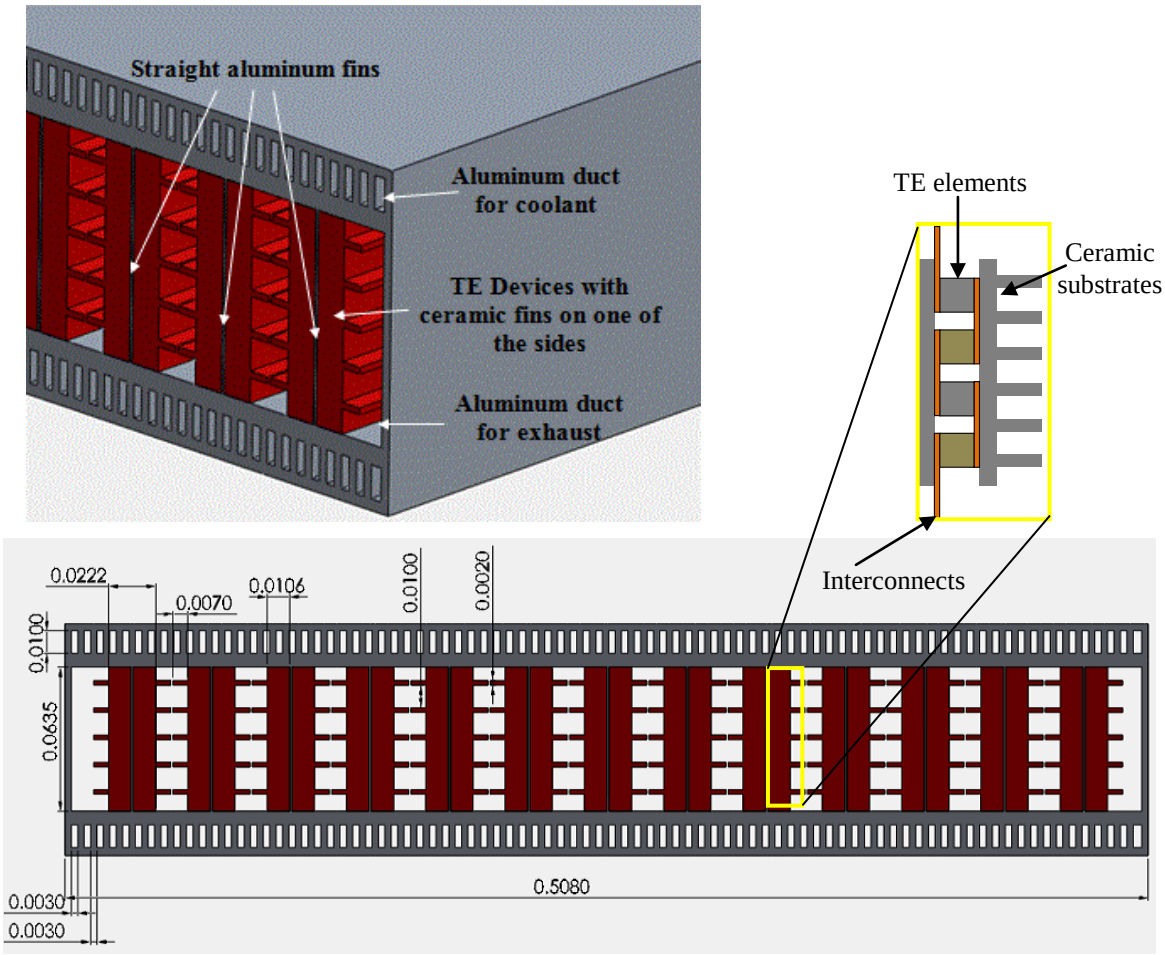


Fig 17: New thermoelectric waste heat recovery system proposed in this study for automotive applications (isometric and front view). TEG modules (red) are attached on the aluminum fins which are a part of the cooling system.TEG modules would need to be sealed against exhaust gases at the inlet and exhaust. All dimensions are in meters. Heat exchanger length is 0.508 m.

The new heat exchanger consists of straight fins in the exhaust side. TEG modules are attached to the fins such that the modules are in direct contact with the hot gases on one side and with the aluminum fins on the other side (cold side). A large fraction of the wetted area inside the exhaust side is covered with TE modules.

The results for optimized designs for both the conventional and the new heat exchanger are shown in Table 5. Both the results are presented for exhaust temperature of 800 K at the inlet and a coolant temperature of 300 K at the inlet. The optimized geometrical parameters are shown in Fig. 13 and Fig. 14 for the conventional and the new design, respectively. Both the results are for TEG modules with a leg height of 0.0076 m, fill fraction 18%, n-/p-type area ratio 0.7, and a current of 1.28 A. The new system configuration shows an improvement of 40% over the conventional system for the same volume of system package. The new system achieves this by extracting more heat from the hot exhaust gas compared to the old design. This analysis accounts for the pumping power required for coolant flow and is therefore based on net power output. It is also evident that pumping power required in the case of new design is much larger compared to the old design because of smaller flow area encountered by the exhaust flow in the new design.

Table 5: Modeling results for the conventional and the new waste heat recovery system configuration (these results reflect optimized parameters for maximum power output for each case for a given TE device geometry).The convection area and the pressure drop between the two designs are different because of the difference in fin arrangements.

Parameter	Conventional system configuration	New (proposed) system configuration
Net Power (W)	572	805
Raw TEG Power (W)	587	926
Pumping Power (W)	15	121
Total Heat Extracted (kW)	12.6	20.6
TE Device Efficiency (%)	4.67	4.5

Chapter 5: TE Device Optimization Study

As explained previously, four parameters, leg length, fill fraction, leg area ratio, and load resistance were considered for optimization of a TEG. Boundary conditions used for this modeling study are listed in Table 6. To simulate a typical finned heat exchanger geometry, the overall heat transfer coefficients averaged in the streamwise direction for both the hot and cold sides of a conventional straight fin heat exchanger (obtained from the system level model) were used. These numbers were reported for a heat exchanger installed in the exhaust of a Cummins 6.7 L diesel engine.

Table 6: Specified boundary conditions for this study. These are typical values for coolant and exhaust averaged in the stream-wise direction

Parameter	Value
Hot side convection temperature	600-800 K
Cold side convection temperature	300 K
Hot side overall convection coefficient	2.0 kW/m^2
Cold side overall convection coefficient	8.0 kW/m^2

All four parameters require optimization to achieve the maximum power flux. The results showed that the sensitivities could be very different among the parameters, however, it was found that the maximum power flux remained in a narrow range of leg area ratio between n-type and p-type legs even as the other three parameters varied widely. This is because the temperature dependency of the TE properties exhibited a similar trend for both the n-type and p-type materials. For the boundary conditions provided in Table 6, the leg area ratio between n-type and p-type legs was fixed at 0.42 based on the optimization results. In contrast, for conventional bismuth telluride TE materials, the leg area ratio was found to be roughly unity. An analytical value for this ratio can be estimated using the equation [44],

$$\frac{A_n}{A_p} = \sqrt{\frac{\rho_n k_p}{\rho_p k_n}} \tag{35}$$

This equation estimates an optimum leg area ratio of 0.46 between n-type and p-type legs. However, this optimal area ratio is based on an averaged values of ρ and k for the given hot and cold side temperatures. For the given boundary conditions and range of operating temperatures, theoretical analysis slightly over-predicts the optimum leg area ratio by approximately 10%. If the TE properties are averaged over a range of temperature and assumed to be independent of temperature, the numerical model is expected to predict the same area ratio as the theoretical value. However, the theoretical value is based on an assumption that the load resistance is exactly equal to the internal resistance of the module, which is not true for the numerical model.

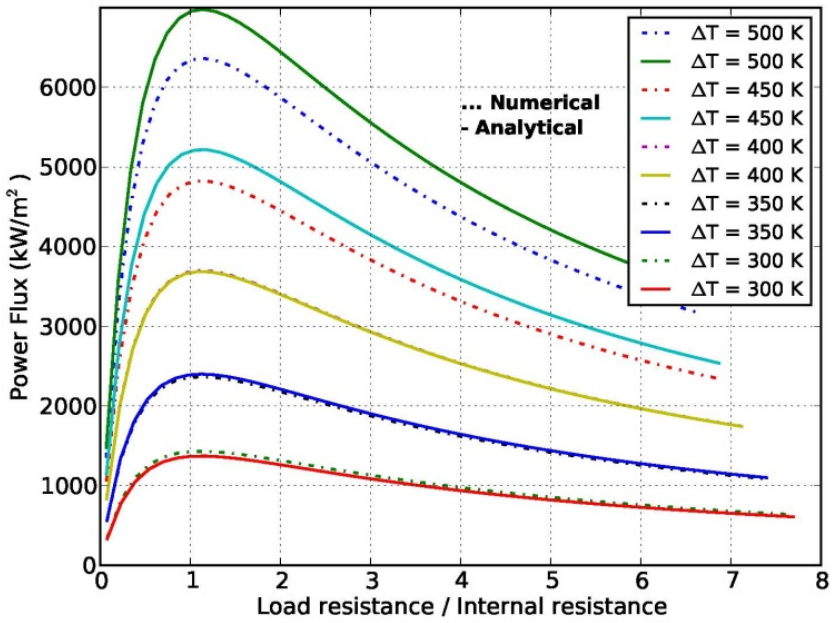


Fig. 18: The effect of load resistance on the power produced per unit area of the TEG module. Maximum power flux is produced when the electrical load resistance is equal to the internal resistance. Cold side convection boundary condition is kept constant at 300 K.

The variation of net power flux output $P_{flux,out}$ for a TEG module at different values of electrical load resistance is shown in Fig. 18. The x-axis has been normalized

by the internal resistance of the module. Maximum power flux is achieved at a load resistance nearly equal to the internal resistance of the module. For load resistance below this optimum value, there is a steep drop in the power output as load resistance decreases. The decrease is more gradual with increasing load resistance; this behavior would have implications for specifying load resistance for a practical system where it would be desirable for the available power to be less sensitive to load resistance. Without accounting for the temperature dependence of the TE properties, an analytical expression representing power flux output as a function of resistance ratio, m', is [44],

$$P_{flux,out} = \frac{P_{out}}{Area} = \frac{\left[\left(\alpha_p - \alpha_n\right)\left(T_{Hot,leg} - T_{Cold,leg}\right)\right]^2 m'}{(1+m')^2 R_{internal}} \frac{1}{Area} \tag{36}$$

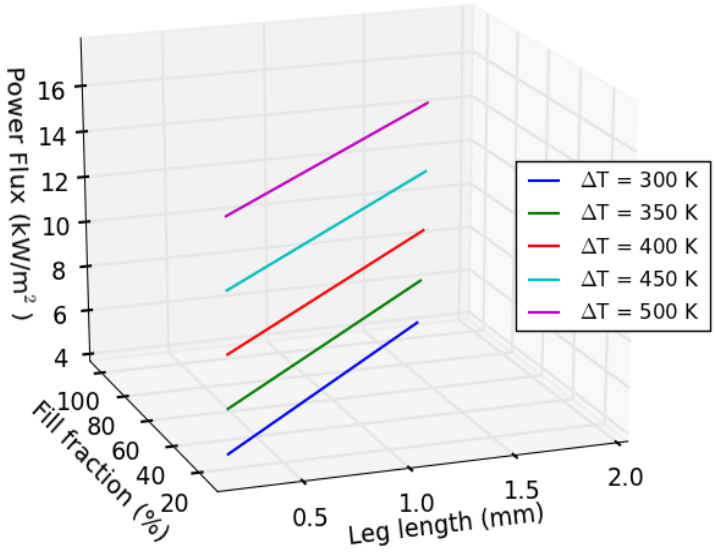
For a given temperature difference, length, and cross sectional area of TE legs, the power flux output is plotted against electrical resistance ratio for both based on this analytical expression and for the model, in Fig. 18.

It is clear from Fig. 18 that the theoretical and numerical results follow the same trends. However, the analytical solution slightly over-predicts the power flux compared to the numerical solution. This is because the numerical solution uses temperature dependent TE properties as opposed to temperature averaged TE properties used in theoretical analysis. This discrepancy can be attributed to the non-linear dependence of ZT on temperature for the silicides under consideration. For temperature differences less than 400 K, the differences in predicted power flux between the analytical and numerical model are small, but they increase with increasing temperature difference such that the analytical model may over-predict power by more than 10% at 500 K temperature difference.

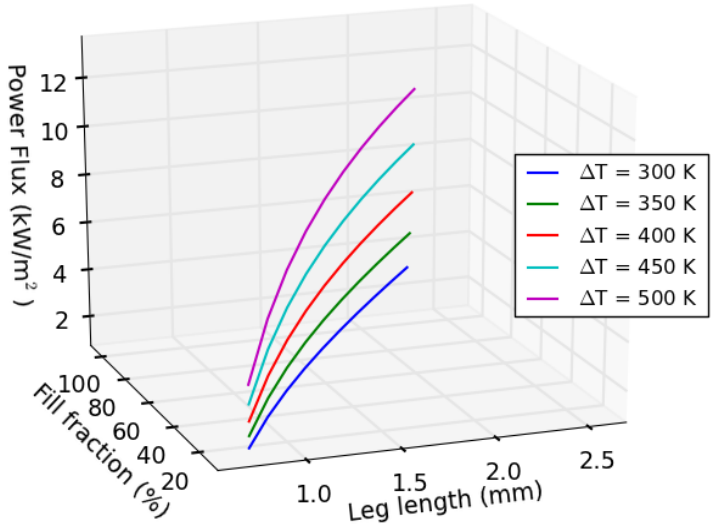
Two different cases regarding the thermal circuit shown in Fig. 5 have been considered in the present study. In the first case, the model assumption is that the thermal contact resistance between the TE elements and the heat exchangers is negligible. This is, an ideal case where thermal contact resistance between the two has not been considered (R_2 = 0 in Fig. 5). For the second case, the thermal resistance due to the contact resistance between various constituting layers such as the metal interconnects and

electrically insulating ceramic substrates has been included in the model. Contact resistance has been found to be a strong function of operating temperature, clamping pressure, surface finish, and material composition, among other variables [46,47,48]. After a comprehensive literature review, a combined contact resistance in the range of 0.0001 m^2-K/W and 0.00001 m^2-K/W was considered reasonable for modeling purposes in this study (R_2 = 0.0001 - 0.00001 m^2-K/W in Fig. 5).

Modeling results for the two cases are shown in Fig. 19(a) and 19(b), respectively. For the first case, which characterizes an ideal scenario where the thermal resistance between the TE elements and the exchangers is negligible, the optimization results are shown in Fig. 19(a). A very important observation made in this case is that there are multiple solutions in the geometrical optimization of TEG modules. Various combinations of optimal leg length, fill fraction, leg area ratio, and load resistance (or current) can be achieved, which provide the same theoretical peak power flux, given model assumptions. As discussed earlier, the leg area ratio and the ratio of electrical load resistance to internal resistance for peak power flux output are 0.42 and unity, respectively. For a given temperature difference, each point on the line in the Fig. 19(a) corresponds to the same peak power output per unit area of the TEG module. This plot illustrates that fewer legs, i.e., a small fill fraction-with legs of shorter length have the potential to achieve the same peak power flux output as a greater number of longer legs, when integrated into a heat exchanger. Essentially, a smaller volume of TE material can provide the same power flux, given the geometrical configuration is accurately optimized. This is shown in the Fig. 20 where the volume of TE materials is plotted against fill fraction for optimal cases. For example, the same power flux can be produced with 40 cm^3 of TE material as with 400 cm^3 of material per m^2 heat exchanger area. The relationship between fill fraction and leg length that results in optimal TEG performance presented in this study is consistent with the results of Gomez et.al. [20].



(a) Negligible thermal contact resistance



(b) Contact resistance included (R_2 = 0.0001 m^2-K/W)

Fig. 19: Peak power per unit area for a TEG module plotted against fill fraction and leg length. Each point on a given line represents the maximum power output that can be achieved for the given fill fraction or leg length, while keeping other parameters at optimum. The leg area ratio between the n-type and p-type legs is optimized and fixed at 0.42, and the electrical load resistance for each solution is the same as the internal resistance. Results are shown for a temperature difference range of 300 K - 500 K between the exhaust gas and the coolant.

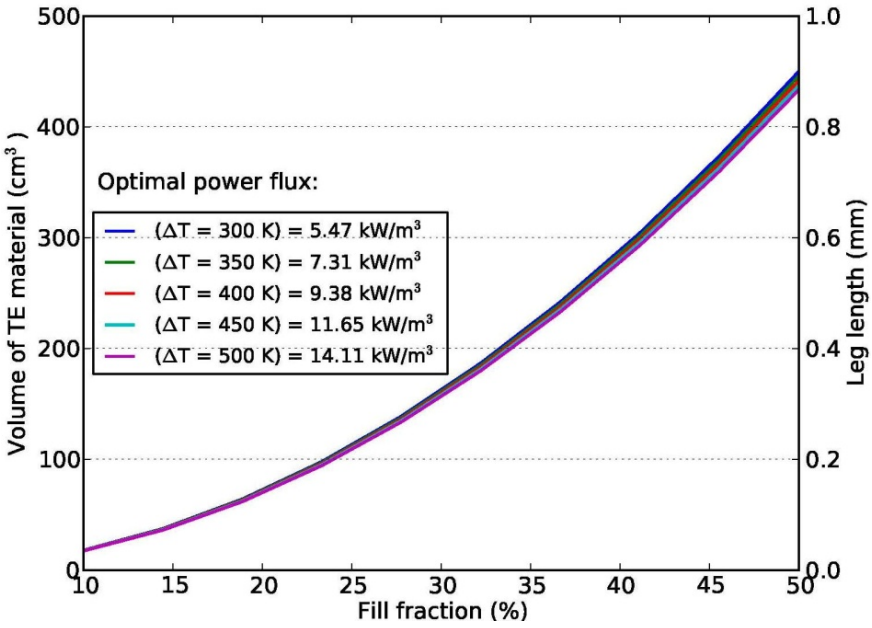


Figure 20: Volume of TE material per m^2 and optimum leg length for optimal power flux vs. Fill fraction at various TE leg temperature differences

For the first case, where contact resistances are neglected, when one of the many possible solutions is selected and further studied in detail, a surface projection of a 3-D design space as shown in Fig. 21 is obtained. It is important to note that these plots refer to a single point along the optimal performance curve in a 3D space (shown in Fig. 19(a), and they should not be confused with the representation of one unique optimal solution. These plots correspond to a constant leg area ratio of 0.42, as well. The figures illustrate that when keeping one of the three parameters - leg length, fill fraction, and current, constant, there is a unique solution for the other two parameters that produces maximum power flux. Deviating from this combination can adversely affect the performance of the TEG module. For instance, increasing or decreasing current by 50% while keeping fill fraction and leg length fixed at optimum value can lower the power flux output by approximately 20%.

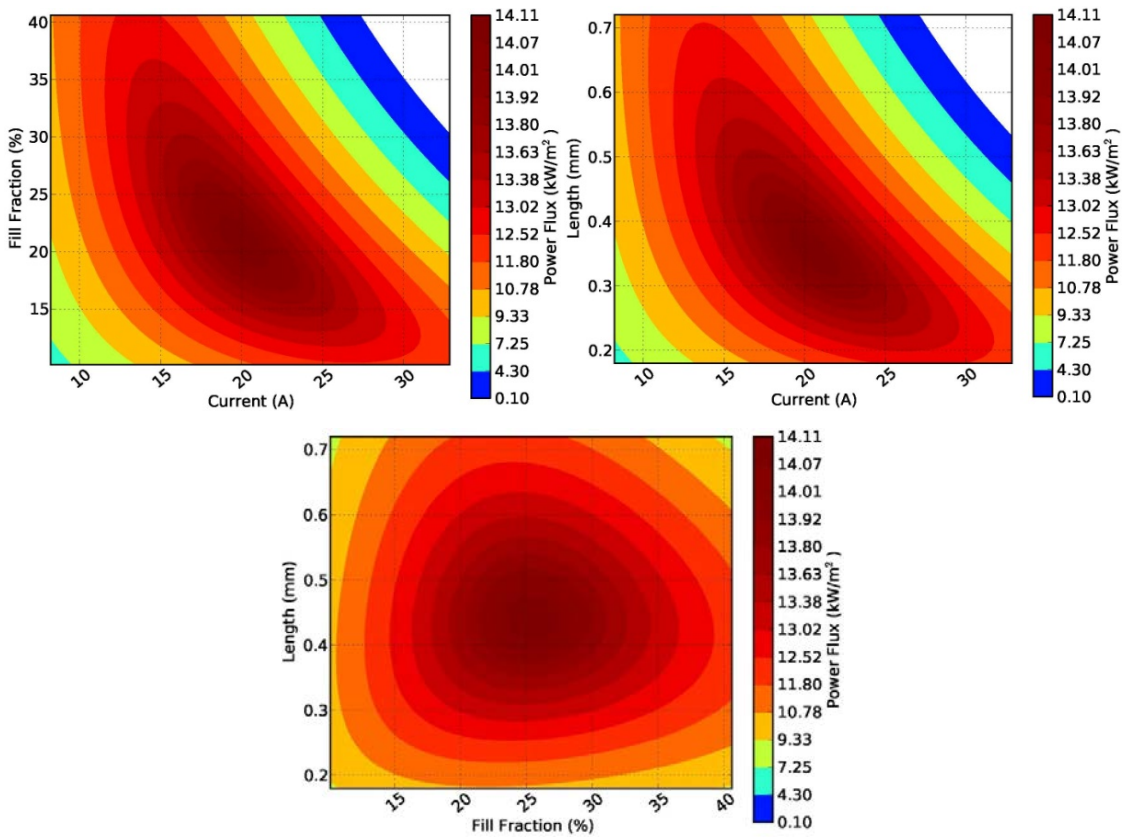


Figure 21: 3D Surface plots showing thermoelectric power flux output vs. (a) current and fill fraction, (b) current and length, and (c) fill fraction and length. n-/p-type leg area ratio is held constant at 0.42. convection boundary conditions are 300 K and 800 K with overall heat transfer coefficients of 8.0 kW/m^2-K and 2.0 kW/m^2-K for the coolant and exhaust, respectively. These figures correspond to an optimized configuration for a fill fraction of 20%. Plots with similar characteristics can be produced for different fill fractions.

For the second case, where thermal contact resistance between the TE elements and the hot and cold side heat exchangers is included in the model, the optimization results are shown in Fig. 19(b). Peak power flux output has been plotted against leg length and fill fraction for various temperature differences between the exhaust and the coolant. The plot shows that the power output increases as fill fraction and leg length are both increased. This shows that a greater number, i.e., larger fill fraction-of longer legs produces greater power output than fewer short legs, when the contact resistance is taken into account. It is desirable to have maximum temperature drop across the TE elements for peak power output. For a given heat flux extracted from the exhaust gas at specified

exhaust and coolant temperatures, the total thermal resistance in between has to stay the same. Increasing the fill fraction, and hence the contact area, leads to smaller contact resistance. This allows for the thermal resistance of the TE elements to increase by increasing their length. The total thermal resistance is still the same, but the temperature drop across the TE elements is larger because of their greater thermal resistance. This conclusion is relevant to the practical application of such TE devices where thermal contact resistance can be a major design parameter.

For exhaust and coolant temperatures of 800 K and 300 K respectively, the peak power output for the case where thermal contact resistance between the TE elements and the heat exchangers is considered negligible is predicted to be 14.1 kW/m^2. For a 50% fill fraction, incorporating a contact resistance of 0.00001 m^2-K/W between the TE elements and the heat exchanger on each sides drops the peak power output by 1.2% to 13.9 kW/m^2, and increasing the contact resistance to 0.0001 m^2-K/W reduces the peak power output by nearly 36%. The dependence of peak power output on thermal contact resistance is shown in Fig. 22.

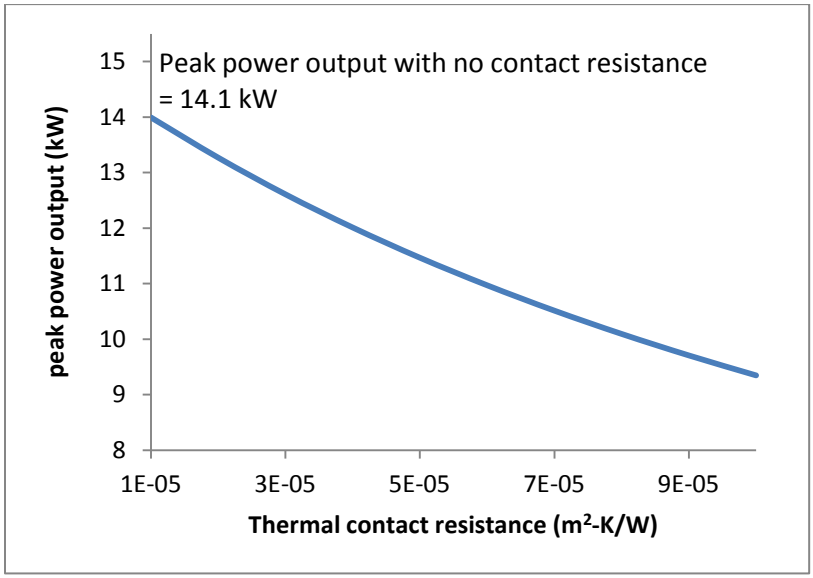


Figure 22: Peak power output as a function of thermal contact resistance in the range of 0.00001-0.0001 m^2-K/W at a fill fraction of 50%. Total system volume is 29.5 L and the exhaust and coolant temperatures are 800 K and 300 K, respectively.

Chapter 5: Conclusions and Recommendations

This study presents numerical and experimental analyses investigating various performance aspects of a thermoelectric waste heat recovery system designed for waste heat recovery from motor vehicles exhaust. It models this system at both the device and system level. A 1-D finite difference numerical model was developed for a system level waste heat recovery configuration using thermoeletcrics. The model included a TE device model and a heat exchanger model, both coupled to produce a system model. The model correctly accounted for temperature dependent properties of TE materials, which is an important feature given the strong dependency of TE properties on temperature. It also accounted for thermal asymmetry between the hot and the cold side resulting from TE energy conversion. The TE device model was developed to be subject to reservoir temperature and heat transfer boundary conditions rather than constant temperature boundary conditions, as observed in the studies in the past. The numerical model was also capable of optimizing various design parameters of the TE device as well as the system model. This was particularly useful in conducting an optimization study of TE device configurations. Finally, this research was performed for Mg and Mn based silicides, which have potential cost advantages with comparable ZT values at higher temperatures relative to conventional TE materials with specific applicability to waste heat recovery systems in vehicles.

The experimental efforts focused mainly on fabricating a TE device using Mg and Mn silicide TE materials. The SPS technique was used to construct the 1^{st} and the 2^{nd} generation TE devices. These devices were used to validate the TE device model for power output. In addition, electrical contact resistance of both devices were experimentally measured to observe the differences between various bonding processes utilized during fabrication. Experimentally measured electrical contact resistance was also incorporated into the numerical model to validate the TE device model. Experimental validation of the system level model was out of scope of this study.

The model was used to optimize TEG module fill fraction, leg length of TE elements, leg area ratio between n- and p- type legs, and load resistance. Two different cases were considered for the numerical TE device model optimization. For the first case, the thermal contact resistance between the TE elements and the heat exchangers, was considered to be negligible. For this case, the numerical model predicted a maximum power generation of 14.1 kW/m^2 for exhaust and coolant temperatures of 800 K and 300 K respectively. For the second case, the contact resistances were incorporated into the model to simulate realistic operating conditions. For this case, the peak power output increased with increasing TE leg length and fill fraction. At 90% fill fraction, the peak power output was 11.18 kW/m^2 for exhaust and coolant temperatures of 800 K and 300 K respectively. For the first case, the optimization study concluded that a smaller number of shorter legs are capable of producing the same power per unit module area as a greater number of longer legs. However, when the contact resistance was incorporated into the model, it was observed that a greater number of longer legs will produce higher power than fewer short legs.

For both cases, optimum leg area ratio between n- and p-type legs was determined to be 0.42 for exhaust gas temperatures of 600-800 K and coolant temperature of 300 K. A load resistance nearly equal to the internal resistance of the module was found to produce maximum power per unit area of the heat exchanger for exhaust and coolant temperatures of 800 K and 300 K. Including thermal contact resistance in the model resulted in a drop in peak power by 1.2% and 36% for contact resistances of 0.00001 m^2-K/W and 0.0001 m^2-K/W, respectively.

A new heat exchanger design was proposed for the waste heat recovery system. A comparative study between a convectional system and the new system, both occupying same package volume and subject to the same boundary conditions was performed. The new system showed a net power output of 805 W, compared to the conventional system configuration, which produced a net power output of 572 W only, for both the TEG systems occupying a volume of 29.5 L each. The new system configuration achieved this by keeping thermoelectric devices in direct contact with the exhaust gases, thus reducing

the thermal contact resistance by half. Also, the new configuration was capable of channeling heat at the rate of 20.6 kW to the TEG modules compared to the old system, which could only manage 12.6 kW. This was done by an increasing the fin area of the heat exchangers. The new design, however, required an increased volume of TEG modules by 63% for the same system volume.

The 1st and 2nd generation TE pair devices were designed and fabricated using Mg and Mn silicides to experimentally validate the model. The numerical model under predicted the power output for the 1st and 2nd generation TE device by an average of 20% and 23%, respectively, over the range of temperatures measured. This discrepancy was attributed to the reading of a lower temperature difference on the TE elements than the actual temperature difference because of geometrical constraints in locating thermocouples and radial heat losses due to convection and radiation. A very large improvement in electrical contact resistance was observed when the SPS technique was used to perform the TE element - Cu electrode bonding compared to using a highly conductive commercial silver paste. The electrical contact resistance for a single TE pair device was measured to be 2.503 ohms and 0.0071 ohms for the 1st and 2nd generation TE devices, respectively. However, one drawback of the SPS process to bond the electrodes to TE samples was a decrease in the Seebeck coefficient to 70% of its original value.

Appendices

Appendix A

Flow property calculation

Thermal properties of air - density, dynamic viscosity, thermal conductivity, and specific heat were calculated as follows [53][54].

Dynamic viscosity, μ, at temperature $T$was calculated using,

$$\mu = \frac{5}{16}\frac{\sqrt{\pi m k_B T}}{\pi d^2}$$

Where, m is the molecular mass, k_B is Boltzmann's constant, and d is the collision diameter of the molecule. For air,

$$m = 28.964\ kg$$

$$d = 3.617\ \dot{A}$$

Prandtl number = Pr = 0.74

Specific heat = $C_{p,air} = 3.653 + 1.337 \times 10^{-3}T^1 + 3.294 \times 10^{-6}T^2 + 1.913 \times 10^{-9}T^3 + 0.2763 \times 10^{-12}T^4$

Knowing the ideal gas density of air ρ, μ, and Pr, we can calculate diffusivity, α , as,

$$\alpha = \frac{\mu}{\rho Pr}$$

which allows us to calculate temperature-dependent thermal conductivity, k, as,

$$k = \alpha \rho C_{p,air}$$

Appendix B

Electrical Resistance Uncertainty Data

The resolution of devices measuring voltage and temperature were 0.001 mV and 0.1 K. The error associated with calculation of the slope of I-V curve is given in table B1.

Table B1: Error associated with linear fit of the data and slope

Probe location	R^2 of linear fit	Slope and slope error
A-B	0.978	0.0016 $\pm$ 5.49e-5
B-C	0.994	0.0059 $\pm$ 1.04e-4
C-D	0.989	0.0047 $\pm$ 1.13e-4
D-E	0.965	0.0017 $\pm$ 7.84e-5
E-F	0.977	0.0008 $\pm$ 3.01e-5
A-F	0.997	0.0015 $\pm$ 1.70e-4

Appendix C

Python Codes

```
        ==================== coolant.py ====================

"""Contains class for coolant side of heat exchanger."""

# In local directory
import functions
reload(functions)
import enhancement
reload(enhancement)


class Coolant(object):
    """
    class for coolant flow

    Methods:

    __init__
    set_fluid_props

    """

    def __init__(self):
        """
        Sets contants and instantiates classes.

        self.enh_lib = enhancement - Used in hx.py

        """

        self.enh_lib = enhancement
        self.enh = None

        self.height = 1.e-2
        # height (m) of coolant duct
        self.mdot = 1.0
        # mass flow rate (kg/s) of coolant
        self.ducts = 2 # number of coolant ducts per hot duct
        self.geometry = 'parallel plates'
```

```
        self.c_p = 4.179
        # Specific heat (kJ/kg*K) of water at 325K
        self.mu = 5.3e-4
        # viscosity of water at 325K (Pa*s), WolframAlpha
        self.k = 0.646e-3
        # thermal conductivity of water at 325K (kW/m*K) through
        # cooling duct
        self.Pr = (7.01 + 5.43)/2 # Prandtl # of water from
Engineering
        # Toolbox
        self.rho = 1000.
        # density (kg/m**3) of water
        self.Nu_coeff = 0.023
        self.enthalpy0 = 113.25
        # enthalpy (kJ/kg) of coolant at restricted dead state
        self.entropy0 = 0.437
        # entropy (kJ/kg*K) of coolant at restricted dead state

        functions.bind_functions(self)

    def set_fluid_props(self):

        """Sets fluid properties needed for set_flow."""

        self.nu = self.mu / self.rho


         ==================== engine.py ====================


"""Contains Engine class used to determine attributes of engine """

# User defined modules
import properties as prop

class Engine(object):

    """Class definition for engine object.

    Methods:

    self.set_mdot_charge

    """

    def __init__(self,**kwargs):

        """Sets constants

        Methods:
```

```
        self.air.set_TempPress_dependents()

        Instantiated in hx.py in HX class

        """

        if 'RPM' in kwargs:
            self.RPM = kwargs['kwargs']
        else:
            self.RPM = 2000. # engine speed (RPM)
        if 'torque' in kwargs:
            self.torque = kwargs['torque']
        else:
            self.torque = 300. # brake torque (lb-ft)
        self.displacement = 6.7e-3 # engine swept displacement
(m**3)
        self.cylinders = 6. # number of cylinders
        self.eta_V = 1. # volumetric efficiency of engine. Can
exceed unity for
        # turbo-charged unthrottled engine. Accounts for error
        # in intake manifold pressure estimate.
        self.T_intake = 300. # engine intake temperature (K)
        self.P_intake = 101.325 # pressure (kPa) at intake manifold
        self.air = prop.ideal_gas() # engine working fluid is ideal
gas with
        # the properties of air
        self.air.T = self.T_intake
        self.air.P = self.P_intake
        self.air.set_TempPres_dependents()

    def set_mdot_charge(self):

        """Sets charge mass flow rate."""

        self.mdot_charge =(
            (self.RPM / 2. * self.displacement * self.eta_V *
            self.air.rho) / 60.
            )
        # charge flow (kg/s) in engine


      ==================== enhancement.py ====================


# coding=utf-8
"""Contains classes for modeling convection heat transfer
enhancement."""

# Distribution libraries
import numpy as np
```

```
class IdealFin(object):

    """Class for modeling straight fin.

    Mills, A. F. Heat Transfer. 2nd ed. Prentice Hall, 1998.

    Methods:

    __init__
    set_eta
    set_enh_geometry
    set_h_and_P
    solve_enh

    """

    def __init__(self, flow):

        """Sets constants and things are needed at runtime.  Runs
        set_fin_height and set_area_convection."""

        self.thickness = 1.e-3
        # fin thickness (m)
        self.k = 0.2
        # thermal conductivity (kW / (m * K)) of fin material
        self.spacing = 0.003
        # distance (m) between adjacent fin edges

        self.flow = flow
        self.set_fin_height()

    def set_fin_height(self):

        """Sets fin height based on half of duct height."""

        self.height = self.flow.height / 2
        # height of fin pair such that their tips meet in the
        # middle and are adiabatic.

    def set_area_convection(self):

        """Sets finned and unfinned area for convection."""

        self.flow.area_unfinned = (
            self.flow.width - self.N * self.thickness
            )
        self.flow.area_finned = self.N * self.height * 4.
        #print "Finned area ", self.flow.area_finned

    def set_enh_geometry(self):
```

```
        """Fixes appropriate geometrical parameters.

        """

        self.set_fin_height()

        # self.N = ((self.flow.width / self.spacing - 1.) / (1. +
        # self.thickness / self.spacing))

        self.N = (
            self.flow.width / (self.thickness + self.spacing)
            )

        self.flow.perimeter = (
            2. * (self.spacing + self.flow.height) * (self.N + 1.)
            )
        # perimeter of new duct formed by fins with constant overall
duct width
        self.flow.flow_area = self.spacing * self.flow.height *
(self.N + 1.)
        # flow area (m^2) of new duct formed by fin

        self.flow.D = 4. * self.flow.flow_area / self.flow.perimeter
        # hydraulic diameter (m)
        #print "\nHydraulic diameter is ", self.flow.D

        self.set_area_convection()

    def set_eta(self):

        """Sets fin efficiency and related parameters.

        """

        self.beta = np.sqrt(
            2. * self.flow.h_conv / (self.k * self.thickness)
            )
        # dimensionless fin parameter
        self.xi = self.beta * self.height
        # beta times fin length (self.height)
        self.eta = np.tanh(self.xi) / self.xi
        # fin efficiency

    def set_h_and_P(self):

        """Sets effective heat transfer coefficient and deltaP.

        """

        self.flow.h_unfinned = self.flow.h_conv
```

```
        self.effectiveness = (
            self.eta * 2. * self.height / self.thickness
            )
        self.h_base = self.effectiveness * self.flow.h_conv

        self.flow.h_conv = (
            (self.flow.h_unfinned * self.flow.area_unfinned +
            self.flow.h_unfinned * self.flow.area_finned * self.eta)
/
            (self.flow.area_finned + self.flow.area_unfinned)
            )
        #print "\nh_conv primary is ", self.flow.h_conv

        self.flow.h_conv = (
            self.flow.h_conv * (self.flow.area_finned +
            self.flow.area_unfinned) / self.flow.area_finned
            )
        #print "h_conv is ", self.flow.h_conv

        self.flow.deltaP = (
            self.flow.f * self.flow.perimeter *
self.flow.node_length
            / self.flow.flow_area * (0.5 * self.flow.rho *
            self.flow.velocity ** 2) * 0.001
            )
        # pressure drop (kPa)

    def solve_enh(self):

        """Runs all the other methods that need to run.

        Methods:

        self.set_enh_geometry
        self.set_eta
        self.set_h_and_P

        """

        self.flow.set_Re_dependents()
        self.flow.h_conv = self.flow.Nu_D * self.flow.k /
self.flow.D
        # coefficient of convection (kW/m^2-K)

        self.set_eta()
        self.set_h_and_P()

class ComplexFin(object):

    """Class for modeling straight fin.
```

```
Mills, A. F. Heat Transfer. 2nd ed. Prentice Hall, 1998.

Methods:

__init__
set_eta
set_enh_geometry
set_h_and_P
solve_enh

"""

def __init__(self, flow):

    """Sets constants and things are needed at runtime.  Runs
    set_fin_height and set_area_convection."""

    # self.smaller_fin_thickness = 2.0e-3
    # self.smaller_fin_spacing = 2.0e-3
    # self.smaller_fin_height = 4.0e-3
    self.smaller_fin_thickness = 2.0e-3
    self.smaller_fin_spacing = 10.0e-3
    self.smaller_fin_height = 7.0e-3
    self.thickness = 1.e-3
    # fin thickness (m)
    self.k = 0.2
    # thermal conductivity (kW / (m * K)) of fin material
    self.spacing = 0.003
    # distance (m) between adjacent fin edges

    self.flow = flow
    self.set_fin_height()

def set_fin_height(self):

    """Sets fin height based on half of duct height."""

    self.height = self.flow.height / 2
    # height of fin pair such that their tips meet in the
    # middle and are adiabatic.

def set_area_convection(self):

    """Sets finned and unfinned area for convection."""

    # # area of the base of the heat exchanger, not the base on
    # # the larger fins
    self.flow.area_unfinned_base = (
        2. * (self.flow.width - self.N * self.thickness)
        )
```

```
        # # finned area of the smaller fins
        self.flow.area_finned_small = (
            self.N * self.n * (self.smaller_fin_thickness + 2. *
            self.smaller_fin_height)
            )

        # # base area of the larger fins
        self.area_unfinned_large = (
            (self.N * 2. * self.flow.height - self.n *
             self.smaller_fin_thickness)
            )

        self.flow.area_unfinned_large = (
            (self.N * 2. * self.flow.height - self.n *
             self.smaller_fin_thickness)
            )

        #print "\ngot here?? "
        self.flow.area_hot_convection = (
            self.N * 2 * self.flow.height
            )

    def set_enh_geometry(self):

        """Fixes appropriate geometrical parameters.

        """

        self.set_fin_height()

        # # number of large fins
        self.N = (
            self.flow.width / (self.thickness + self.spacing)
            )

        # # number of small fins per one large fin
        self.n = (
            2 * self.flow.height / (self.smaller_fin_thickness +
            self.smaller_fin_spacing)
            )

        self.perimeter1 = (
            self.N * self.n * (self.smaller_fin_height * 2. +
            self.smaller_fin_thickness + self.smaller_fin_spacing)
            )
        self.perimeter2 = (
            2. * self.N * self.spacing
            )
        self.flow.perimeter = self.perimeter1 + self.perimeter2
        # perimeter of new duct formed by fins with constant overall
        # duct width
```

```
        self.total_area = (
            self.flow.width * self.flow.height
            )
        self.area1 = (
            self.flow.height * self.thickness * self.N
            )
        self.area2 = (
            self.N * self.n * (self.smaller_fin_height *
        self.smaller_fin_thickness)
            )

        self.flow.flow_area = self.total_area - self.area1 -
self.area2
        # flow area (m^2) of new duct formed by fin

        self.flow.D = 4. * self.flow.flow_area / self.flow.perimeter
        # hydraulic diameter (m)
        #print "\nHydraulic diameter is ", self.flow.D

        self.set_area_convection()

    def set_eta(self):

        """Sets fin efficiency and related parameters.

        """

        self.beta = np.sqrt(
            2. * self.flow.h_conv / (self.k * self.thickness)
            )
        # dimensionless fin parameter
        self.xi = self.beta * self.height
        # beta times fin length (self.height)
        self.eta = np.tanh(self.xi) / self.xi
        # fin efficiency

    def set_eta_small(self):
        """ fin efficiency for the smaller fins """

        self.beta_small = np.sqrt(
            2. * self.flow.h_conv / (self.k *
self.smaller_fin_thickness)
            )
        # dimensionless fin parameter
        self.xi_small = self.beta_small * self.smaller_fin_height
        # beta times fin length (self.height)
        self.eta_small = np.tanh(self.xi_small) / self.xi_small
        # fin efficiency

    def set_h_and_P(self):
```

```
        """Sets effective heat transfer coefficient and deltaP.

        """

        self.flow.h_unfinned = self.flow.h_conv

        #print "eta small is ", self.eta_small
        self.flow.h_conv = (
            (self.flow.h_unfinned * self.flow.area_unfinned_base +
            self.flow.h_unfinned * self.area_unfinned_large *
            self.eta + self.flow.h_unfinned *
            self.flow.area_finned_small * self.eta_small) /
            (self.flow.area_unfinned_base +
            self.area_unfinned_large + self.flow.area_finned_small)
            )
        #print "\nh_conv primary is ", self.flow.h_conv

        self.flow.h_conv = (
            self.flow.h_conv * (self.flow.area_unfinned_base +
            self.flow.area_unfinned_large +
self.flow.area_finned_small) /
            (self.flow.area_hot_convection)
            )
        #print "h_conv is ", self.flow.h_conv

        self.flow.deltaP = (
            self.flow.f * self.flow.perimeter *
self.flow.node_length
            / self.flow.flow_area * (0.5 * self.flow.rho *
            self.flow.velocity ** 2) * 0.001
            )
        # pressure drop (kPa)

    def solve_enh(self):

        """Runs all the other methods that need to run.

        Methods:

        self.set_enh_geometry
        self.set_eta
        self.set_h_and_P

        """

        self.flow.set_Re_dependents()
        self.flow.h_conv = self.flow.Nu_D * self.flow.k /
self.flow.D
        # coefficient of convection (kW/m^2-K)
```

```
        self.set_eta()
        self.set_eta_small()
        self.set_h_and_P()

        ==================== exhaust.py ====================


"""Contains class for exhaust side of heat exchanger"""

# In python directory
import properties as prop
reload(prop)

# In this directory
import functions
reload(functions)
import enhancement
reload(enhancement)


class Exhaust(prop.ideal_gas):

    """Class for engine exhaust in heat exchanger.

    Methods:

    __init__
    set_fluid_props

    """
    def __init__(self):

        """
        Sets a bunch of constants, binds methods, inits parent class

        self.enh_lib = enhancement - Used in hx.py

        Also initializes super class, which is ideal_gas from the
        properties script.  I keep this script in
~/Documents/Python,
        which is part of my python path.

        """

        super(Exhaust, self).__init__()

        self.enh_lib = enhancement
        self.enh = None
        self.T_ref = 300.
        self.P = 101.
        self.height = 1.5e-2
```

```
        self.ducts = 1

        self.Nu_coeff = 0.023

        functions.bind_functions(self)

    def set_fluid_props(self):

        """
        Sets properties needed for set_flow.

        Methods:

        self.set_thermal_props

        """

        self.set_thermal_props()
        self.c_p = self.c_p_air
        self.k = self.k_air
```

`==================== functions.py ====================`

```
"""Contains functions to be used in both exhaust and coolant
modules."""

import numpy as np
import types

def set_flow_geometry(self, width):

    """Sets perimeter, flow area, and hydraulic diameter.

    Inputs:

    width (m)

    """

    self.perimeter = 2.*(self.height + width)
    # wetted perimeter (m) of flow
    self.flow_area = self.height * width
    # cross-section area (m^2) of exhaust flow
    self.D = 4. * self.flow_area / self.perimeter
    # coolant hydraulic diameter (m)

    try:
        self.enh
    except AttributeError:
```

```
        self.enh = None

    if self.enh != None:
        try:
            self.enh.set_enh_geometry()
        except AttributeError:
            pass
        else:
            self.enh.set_enh_geometry()

def set_Re_dependents(self):

    """Sets Nu and f based on Re.

    Methods:

    self.set_Re

    """

    self.set_Re()
    if (self.Re_D > 2300.): # Do these correlations hold for any
tube geometry?
        self.f = 0.078 * self.Re_D**(-1. / 4.) # friction factor for
turbulent
        # flow from Bejan
        self.f = self.f * 1.5  # scaled for parallel plates
according
        # to Bejan Convection Heat Transfer, 3rd ed. Table 3.2

        self.Nu_D = self.Nu_coeff * self.Re_D**(4. /
5.)*self.Pr**(1. / 3.) # Adrian
        # Bejan, Convection Heat Transfer, 3rd ed., Equation 8.30
        self.flow = 'turbulent'
    else:
        # self.Nu_D = 4.36 # Hesselgreaves, compact heat exchangers,
constant T
        # self.f = 16. / self.Re_D
        # self.flow = 'laminar'
        self.Nu_D = 7.54 # Bejan, Convection Heat Transfer, Table
3.2
        # parallel plates with constant T
        self.f = 24. / self.Re_D
        self.flow = 'laminar'

def set_Re(self):

    """Sets Reynolds number based on hydraulic diameter.

    Requiures:
```

```
    self.velocity
    self.D
    self.nu
    self.Re_D = self.velocity * self.D / self.nu

    """
    self.Re_D = self.velocity * self.D / self.nu
    # Reynolds number

def set_flow(self):

    """
    Sets flow parameters for exhaust or coolant instance.

    See exhaust.py and coolant.py

    Methods
    -------
    self.set_fluid_props
    self.set_Re_dependents
    self.enh.solve_enh
    self.set_thermal_props()

    Used in hx.py by hx.HX.set_convection and possibly
    elsewhere."""

    self.set_fluid_props()

    self.C = self.mdot * self.c_p
    # heat capacity of flow (kW/K)
    self.Vdot = self.mdot / self.rho
    # volume flow rate (m^3/s) of exhaust
    self.velocity = self.Vdot / self.flow_area
    # velocity (m/s) of exhaust

    self.set_Re_dependents()
    self.h_conv= self.Nu_D * self.k / self.D
    # coefficient of convection (kW/m^2-K)

    if self.enh == None:
        self.deltaP = (
            self.f * self.perimeter * self.node_length /
            self.flow_area * (0.5 * self.rho * self.velocity ** 2.)
* 0.001
            )
        # pressure drop (kPa)
        print """Something is wrong in set_flow in
        Modules/functions.py if you thought you were using
        enhancement."""
    else:
        self.enh.solve_enh()
```

```
    self.Wdot_pumping = self.Vdot * self.deltaP
    # pumping power (kW)

    self.R_thermal = 1. / self.h_conv
    # thermal resistance (m^2-K/kW) of exhaust

def set_enhancement(self, enh_type):

    """For some enhancement strategies, this is useful for
    instantiating the instance because it can pass the exhaust or
flow
    instance to the enhancment instance."""

    if enh_type == 'IdealFin':
        self.enh = self.enh_lib.IdealFin(self)

    elif enh_type == 'IdealFin2':
        self.enh = self.enh_lib.IdealFin2(self)

    elif enh_type == 'OffsetStripFin':
        self.enh = self.enh_lib.OffsetStripFin(self)

    elif enh_type == 'CircularDuct':
        self.enh = self.enh_lib.CircularDuct(self)

    elif enh_type == 'RectangularDuct':
        self.enh = self.enh_lib.RectangularDuct(self)

    elif enh_type == 'IdealFinCoolant':
        self.enh = self.enh_lib.IdealFinCoolant(self)

    elif enh_type == 'ComplexFin':
        self.enh = self.enh_lib.ComplexFin(self)

    else:
        print "Error in enh_type specification."
        print "Possible options are:"
        print "IdealFin"
        print "IdealFinCoolant"
        print "ComplexFin"

    return self.enh

def bind_functions(self):
    """Binds functions used by both coolant and exhaust."""

    self.set_flow_geometry = (
        types.MethodType(set_flow_geometry, self)
        )
    self.set_Re = (
```

```
        types.MethodType(set_Re, self)
        )
    self.set_Re_dependents = (
        types.MethodType(set_Re_dependents, self)
        )
    self.set_flow = (
        types.MethodType(set_flow, self)
        )
    self.set_enhancement = (
        types.MethodType(set_enhancement, self)
        )
```

========== hx.py ==========

```
# coding=utf-8
"""
Script defining HX class.

"""

# Distribution Modules
import time
import numpy as np
import operator
from scipy.optimize import fmin  # _l_bfgs_b

# User Defined Modules
# In this directory
import engine
import te_pair
reload(te_pair)
import exhaust
reload(exhaust)
import coolant
reload(coolant)
import platewall
reload(platewall)

class Dimension(object):
    """Class for hx attribute containing physical dimensions.  This
is
    used on an ad hoc basis."""
    pass

class HX(object):

    """Class definition for heat exchanger.
    """

    def __init__(self):

        """Sets several attributes, including instance attributes.
```

```
        Instance attributes

        self.cool = coolant.Coolant()
        self.exh = exhaust.Exhaust()
        self.te_pair = te_pair.TE_Pair()
        self.plate = platewall.PlateWall()
        self.cummins = engine.Engine()

        Methods:

        self.fix_geometry

        """

        self.R_extra = 0.

        self.R_interconnect = 0.00075 # (m^2*K/kW)
        # Resistance of copper interconnect assuming a thickness of
        # 0.3 mm (Ref: Hori, Y., D. Kusano, T. Ito, and
        # K. Izumi. “Analysis on Thermo-mechanical Stress of
        # Thermoelectric Module.” In Thermoelectrics 1999.
Eighteenth
        # International Conference On, 328 –331, 1999), where
        # k_interconnect = 400 W/(m-K)

        self.R_substrate = 0.005 # (m^2*K/kW)
        # resistance of ceramic substrate(AlN) 1 mmm thick (Hori,
Y.,
        # D. Kusano, T. Ito, and K. Izumi. “Analysis on
        # Thermo-mechanical Stress of Thermoelectric Module.” In
        # Thermoelectrics, 1999. Eighteenth International Conference
        # On, 328 –331, 1999.), based on k_ceramic = 200 W/(m-K)
        # obtained from Thermoelectrics Handbook.

        # self.R_contact = 0.00003 # (m^2*K/kW)
        # Thermal contact resistance for all three contacts
estimated
        # using alumina/copper contact resistance extracted from
        # Gundrum, Bryan C., David G. Cahill, and Robert
        # S. Averback. “Thermal Conductance of Metal-metal
        # Interfaces.” Physical Review B 72, no. 24 (December 30,
        # 2005): 245426.

        # self.R_contact = 0.8322 # (m^2*K/kW)
        # thermal contact resistance (m^2*K/kW) for plate/substrate,
        # substrate/interconnect, and interconnect/TE leg interfaces
        # all combined. All estimated (at 450 K) based on AlN/Cu
        # contact resistance extracted from Shi, Ling, Gang Wu,
        # Hui-ling Wang, and Xin-ming Yu. “Interfacial Thermal
Contact
```

```
        # Resistance Between Aluminum Nitride and Copper at
Cryogenic
        # Temperature.” Heat and Mass Transfer 48, no. 6 (2012):
        # 999–1004.

        self.R_contact = 0.05

        self.dimension = Dimension()
        self.dimension.width = 0.508
        # width (cm*10**-2) of HX duct. This model treats duct as
        # parallel plates for simpler modeling.
        self.dimension.length = 0.508
        # length (m) of HX duct
        self.nodes = 25
        # number of nodes for numerical heat transfer model
        self.x0 = np.array([.7, 0.02, 0.001, 4.])
        self.xb = [(0.5, 2.), (0., 1.), (1.e-4, 20.e-3), (0.1,
None)]
        # initial guess and bounds for x where entries are N/P area,
        # fill fraction, leg length (m), and current (A)
        self.xmin_file = 'xmin'
        self.T0 = 300.
        # temperature (K) at restricted dead state
        self.equal_width = True

        self.apar_list = [
            ['self', 'te_pair', 'leg_area_ratio'],
            ['self', 'te_pair', 'fill_fraction'],
            ['self', 'te_pair', 'length'],
            ['self', 'te_pair', 'I']
            ]
        # list of strings used to construct names of attributes to
be
        # optimized

        # initialization of instance attributes
        self.cool = coolant.Coolant()
        self.exh = exhaust.Exhaust()
        self.te_pair = te_pair.TE_Pair()
        self.plate = platewall.PlateWall()
        self.cummins = engine.Engine()

        self.arrangement = 'single'
        self.fix_geometry()

    def init_arrays(self):

        """Initializes arrays for storing node values."""

        self.Qdot_nodes = np.zeros(self.nodes)
```

```
        self.exh.Vdot_nodes = np.zeros(self.nodes)

        self.exh.T_nodes = np.zeros(self.nodes)
        self.exh.h_nodes = np.zeros(self.nodes)
        self.exh.f_nodes = np.zeros(self.nodes)
        self.exh.deltaP_nodes = np.zeros(self.nodes)
        self.exh.Wdot_nodes = np.zeros(self.nodes)
        self.exh.Nu_nodes = np.zeros(self.nodes)
        self.exh.c_p_nodes = np.zeros(self.nodes)
        self.exh.entropy_nodes = np.zeros(self.nodes)
        self.exh.enthalpy_nodes = np.zeros(self.nodes)
        self.exh.velocity_nodes = np.zeros(self.nodes)
        self.exh.rho_nodes = np.zeros(self.nodes)
        self.exh.Re_nodes = np.zeros(self.nodes)

        self.cool.T_nodes = np.zeros(self.nodes)
        self.cool.entropy_nodes = np.zeros(self.nodes)
        self.cool.enthalpy_nodes = np.zeros(self.nodes)
        self.cool.deltaP_nodes = np.zeros(self.nodes)
        self.cool.Wdot_nodes = np.zeros(self.nodes)

        self.U_hot_nodes = np.zeros(self.nodes)
        self.U_cold_nodes = np.zeros(self.nodes)

        self.te_pair.q_h_conv_nodes = np.zeros(self.nodes)
        self.te_pair.q_c_conv_nodes = np.zeros(self.nodes)
        self.te_pair.q_h_nodes = np.zeros(self.nodes)
        self.te_pair.q_c_nodes = np.zeros(self.nodes)
        self.te_pair.error_nodes = np.zeros([3, self.nodes])
        self.te_pair.T_c_nodes = np.zeros(self.nodes)
        self.te_pair.T_h_nodes = np.zeros(self.nodes)
        self.te_pair.h_nodes = np.zeros(self.nodes)
        self.te_pair.power_nodes = np.zeros(self.nodes)
        self.te_pair.power_nodes_check = np.zeros(self.nodes)
        self.te_pair.eta_nodes = np.zeros(self.nodes)

        #============================================================
        self.te_pair.Vs_nodes = np.zeros(self.nodes)
        self.te_pair.V_nodes = np.zeros(self.nodes)
        self.te_pair.R_internal_nodes = np.zeros(self.nodes)
        self.te_pair.R_load_nodes = np.zeros(self.nodes)
        #============================================================

    def setup(self):

        """Sets attributes that must be defined before running
model.

        Methods:

        self.set_mdot_charge
```

```
        self.set_constants

        Useful for terminal.  Not necessary elsewhere.

        """

        self.exh.T = 800.
        self.cool.T = 300.
        self.set_mdot_charge()
        self.set_constants()

    def set_constants(self):

        """Sets constants used at the HX level.

        Methods:

        self.fix_geometry
        self.exh.set_flow_geometry
        self.cool.set_flow_geometry

        """
        self.set_fin_geometry()

        self.x = np.linspace(0, self.dimension.length, self.nodes)
        self.node_length = self.dimension.length / self.nodes
        # length (m) of each node
        self.area = (
            self.node_length * self.dimension.width *
self.cool.ducts
            )
        self.fins = (
            self.dimension.width / (self.exh.enh.spacing +
                                    self.exh.enh.thickness)
            )
        self.area_TE = (self.fins * 2. * self.exh.height *
self.node_length)

        self.te_pair.set_constants()
        self.leg_pairs = self.area_TE / self.te_pair.area
        self.x_dim = np.arange(self.node_length / 2,
self.dimension.length +
        self.node_length / 2, self.node_length)
        # x coordinate (m)
        self.fix_geometry()

        self.exh.set_flow_geometry(self.exh.width)
        self.cool.set_flow_geometry(self.cool.width)

    def fix_geometry(self):
```

```
        """Matches geometry of ducts.

        Makes sure that common geometry like width and length is the
        same between exh, cool, and the overal heat exchanger.

        """

        if self.equal_width == True:
            self.exh.width = self.dimension.width
            self.cool.width = self.dimension.width
        self.cool.length = self.dimension.length
        self.exh.length = self.dimension.length

    def set_fin_geometry(self):
        """Fixes effective fin thickness based on TE module
thickness"""

        self.added_thickness = 1.5e-3 # substrate and copper
        # thickness, etc

        self.module_thickness = (
            2 * (2. * self.added_thickness + self.te_pair.length)
            )

        self.exh.enh.thickness = (
            self.exh_fin_thickness + self.module_thickness
            )

    def set_mdot_charge(self):

        """Sets exhaust mass flow rate.

        Methods:

        self.cummins.set_mdot_charge

        Eventually, this should be a function of speed, load, and
EGR
        fraction.  Also, it should come from experimental data.
Also,
        it should probably go within the exhaust module.

        """

        self.cummins.set_mdot_charge()
        # mass flow rate (kg/s) of exhaust
        self.exh.mdot = self.cummins.mdot_charge

    def set_convection(self):

        """Sets values for convection coefficients.
```

```
        Methods:

        self.exh.set_flow
        self.cool.set_flow

        """

        self.exh.set_flow()

        self.cool.set_flow()
        #print "h_conv cool is ", self.cool.h_conv


        self.cool.h_conv = (
            self.cool.h_conv * (self.cool.area_finned +
                                self.cool.area_unfinned) /
(self.exh.area_hot_convection)
            )

        self.cool.R_thermal = (1. / self.cool.h_conv)

        self.U_hot = ((self.exh.R_thermal + self.R_parasitic_hot) **
-1)
        self.U_cold = ((self.cool.R_thermal + self.R_parasitic_cold)
        ** -1)

    def solve_node(self, i):

        """Solves for performance of streamwise slice of HX.

        Methods:

        self.set_convection
        self.te_pair.solve_te_pair

        """

        self.te_pair.T_h_conv = self.exh.T
        self.te_pair.T_c_conv = self.cool.T

        self.set_convection()

        if i == 0:
            self.te_pair.T_h = self.exh.T
            # guess at hot side TEM temperature (K)
            self.te_pair.T_c = self.cool.T
            # guess at cold side tem temperature (K)

        self.te_pair.U_hot = self.U_hot
        self.te_pair.U_cold = self.U_cold
```

```
        self.te_pair.solve_te_pair()
        self.q_h = self.te_pair.q_h
        self.q_c = self.te_pair.q_c

        self.Qdot_node = self.q_h * self.area_TE
        # heat transfer on hot side of node, positive values
indicates
        # heat transfer from hot to cold
        # assumin all surface areas have been occupied by TE modules

    def solve_hx(self, ** kwargs):
        # solve parallel flow heat exchanger

        """Solves for performance of all stream-wise nodes.

        Methods:

        self.init_arrays
        self.set_constants
        self.solve_node
        self.store_node_values
        self.set_availability

        """
        self.init_arrays()
        self.set_constants()

        self.R_parasitic_hot = (
            self.R_interconnect + self.R_substrate + self.R_extra
            )

        self.R_parasitic_cold = (
            self.plate.R_thermal + self.R_interconnect +
            self.R_substrate + self.R_contact + self.R_extra
            )
        # print "\nPlate resistance ", self.plate.R_thermal
        # R_parasitic (m^2-K/kW) includes plate resistance from
module
        # platewall, resistance of interconnect and ceramic
substrate
        # and all the contact resistances

        self.exh.node_length = self.node_length
        self.exh.T = self.exh.T_inlet
        # T_inlet and T_outlet correspond to the temperatures going
        # into and out of the heat exchanger.
        if self.type == 'parallel':
            self.cool.T = self.cool.T_inlet
        elif self.type == 'counter':
            self.cool.T = self.cool.T_outlet
```

```
        self.cool.node_length = self.node_length

        # for loop iterates of nodes of HX in streamwise direction
        for i in np.arange(self.nodes):
            self.solve_node(i)
            self.store_node_values(i)

            # redefining temperatures (K) for next node
            self.exh.T = (self.exh.T - self.te_pair.q_h *
self.area_TE /
                self.exh.C)
            if self.type == 'parallel':
                self.cool.T = (self.cool.T + self.te_pair.q_c *
self.area
                    / self.cool.C)
            elif self.type == 'counter':
                self.cool.T = (self.cool.T - self.te_pair.q_c *
self.area
                    / self.cool.C)

        # defining HX outlet/inlet temperatures (K)
        self.exh.T_outlet = self.exh.T
        if self.type == 'parallel':
            self.cool.T_outlet = self.cool.T
        elif self.type == 'counter':
            self.cool.T_inlet = self.cool.T

        self.Qdot_total = self.Qdot_nodes.sum()
        self.effectiveness = (self.Qdot_total / (self.exh.C *
        (self.exh.T_inlet - self.cool.T_inlet)))
        # heat exchanger effectiveness
        self.te_pair.Vs_nodes_total = self.te_pair.Vs_nodes.sum()
        self.te_pair.V_nodes_total = self.te_pair.V_nodes.sum()
        self.te_pair.R_internal_nodes_total = (
            self.te_pair.R_internal_nodes.sum()
            )
        self.te_pair.R_load_nodes_total = (
            self.te_pair.R_load_nodes.sum()
            )

        self.te_pair.power_total = self.te_pair.power_nodes.sum()
        # total TE power output (kW)

        self.exh.Wdot_total = self.exh.Wdot_nodes.sum()
        self.cool.Wdot_total = self.cool.Wdot_nodes.sum()
        self.Wdot_pumping = (self.exh.Wdot_total +
                             self.cool.Wdot_total)
        # total pumping power requirement (kW)

        self.exh.deltaP_total = self.exh.deltaP_nodes.sum()
        self.cool.deltaP_total = self.cool.deltaP_nodes.sum()
```

```
        self.power_net = (self.te_pair.power_total -
        self.Wdot_pumping)

        self.set_availability()

    def store_node_values(self, i):

        """Stores values of parameters of interest in node i.

        This should eventually also store the node valuves for T, q,
        and material properties in the te legs.
        """

        self.Qdot_nodes[i] = self.Qdot_node
        # storing node hot side heat transfer in array

        self.te_pair.q_h_conv_nodes[i] = self.q_h
        self.te_pair.q_c_conv_nodes[i] = self.q_c
        self.te_pair.q_h_nodes[i] = self.te_pair.q_h
        self.te_pair.q_c_nodes[i] = self.te_pair.q_c
        self.te_pair.error_nodes[:, i] = self.te_pair.error
        self.te_pair.T_h_nodes[i] = self.te_pair.T_h
        self.te_pair.T_c_nodes[i] = self.te_pair.T_c
        self.te_pair.power_nodes[i] = self.te_pair.P *
self.leg_pairs
        self.te_pair.Vs_nodes[i] = self.te_pair.Vs * self.leg_pairs
        self.te_pair.V_nodes[i] = self.te_pair.V * self.leg_pairs
        self.te_pair.R_internal_nodes[i] = (
            self.te_pair.R_internal * self.leg_pairs
            )
        self.te_pair.R_load_nodes[i] = (
            self.te_pair.R_load * self.leg_pairs
            )

        self.te_pair.power_nodes_check[i] = (
            self.te_pair.P_flux * self.area
            )
        self.te_pair.eta_nodes[i] = self.te_pair.eta
        self.te_pair.h_nodes[i] = self.te_pair.h_eff

        self.exh.T_nodes[i] = self.exh.T
        self.exh.Vdot_nodes[i] = self.exh.Vdot
        self.exh.f_nodes[i] = self.exh.f
        self.exh.deltaP_nodes[i] = self.exh.deltaP
        self.exh.Wdot_nodes[i] = self.exh.Wdot_pumping
        self.exh.Nu_nodes[i] = self.exh.Nu_D
        self.exh.c_p_nodes[i] = self.exh.c_p
        self.exh.h_nodes[i] = self.exh.h_conv
        self.exh.velocity_nodes[i] = self.exh.velocity
        self.exh.entropy_nodes[i] = self.exh.entropy
```

```
        self.exh.enthalpy_nodes[i] = self.exh.enthalpy
        self.exh.rho_nodes[i] = self.exh.rho
        self.exh.Re_nodes[i] = self.exh.Re_D

        self.cool.T_nodes[i] = self.cool.T
        self.cool.deltaP_nodes[i] = self.cool.deltaP
        self.cool.Wdot_nodes[i] = self.cool.Wdot_pumping

        self.U_hot_nodes[i] = self.U_hot
        self.U_cold_nodes[i] = self.U_cold


    def get_minpar(self, apar):

        """ standalone optimization """
        self.opt_iter = self.opt_iter + 1
        if self.opt_iter % 1 == 0:
            print "\noptimizaton iteration", self.opt_iter
            print "leg length =", self.te_pair.length, "m"
            print "fill fraction =", self.te_pair.fill_fraction *
100., "%"
            print "current =", self.te_pair.I, "A"
            print "area ratio =", self.te_pair.leg_area_ratio
            print "raw power", self.te_pair.power_total * 1000., "W"

        apar = np.array(apar)

        # self.length = apar[0]
        # self.fill_fraction = apar[1]
        # self.leg_area_ratio = apar[2]
        # self.R_desired = apar[3]

        self.te_pair.length = apar[0]
        self.te_pair.leg_area_ratio = apar[1]
        self.te_pair.I = apar[2]
        self.te_pair.fill_fraction = apar[3]

        self.te_pair.set_leg_areas()
        self.set_constants()

        self.solve_hx()

        if (apar <= 0.).any():
            minpar = np.abs(self.te_pair.power_total) ** 3. + 100
            print "Encountered impossible value."

        else:
            minpar = - self.te_pair.power_total

        return minpar
```

```
    def optimize(self):

        """ standalone optimization """
        time.clock()

        self.opt_iter = 0

        self.x0 = (
            np.array([self.te_pair.length,
            self.te_pair.leg_area_ratio, self.te_pair.I,
self.te_pair.fill_fraction])
            )

        from scipy.optimize import fmin

        self.xmin = fmin(self.get_minpar, self.x0)

        t1 = time.clock()

        print '\n'

        print "Optimized parameters:"
        print "leg length =", self.te_pair.length, "m"
        print "fill fraction =", self.te_pair.fill_fraction * 100.,
"%"
        print "current =", self.te_pair.I, "A"
        print "area ratio =", self.te_pair.leg_area_ratio

        print "\npower net:", self.te_pair.power_total * 1000., 'W'

        print """Elapsed time solving xmin1 =""", t1

    def save_opt_par(self, opt_par_dir):
        """Saves parameters found by optimize."""

        if self.opt_iter == 0:
            print """\nError. Script must run the function optimize
            before running save_opt_par."""

        for i in range(self.x0.size):
            varname = '.'.join(self.apar_list[i][1:])
            varval = (
                operator.attrgetter(varname)(self)
                )
            np.save(opt_par_dir + varname, varval)



          ==================== leg.py ====================
# Distribution modules
```

```
import types
import numpy as np
from scipy.integrate import odeint
from numpy.testing import assert_approx_equal
from scipy.optimize import fsolve

# User defined modules
import mat_prop
reload(mat_prop)

class Leg(object):

    """ """

    def __init__(self):

        """ """
        self.I = 0.5
        self.nodes = 10
        self.length = 1.e-3
        self.area = (3.e-3) ** 2.
        self.set_constants()

        self.import_raw_property_data = (
            types.MethodType(mat_prop.import_raw_property_data,
self)
            )
        self.set_properties_v_temp = (
            types.MethodType(mat_prop.set_properties_v_temp, self)
            )
        self.set_TEproperties = (
            types.MethodType(mat_prop.set_TEproperties, self)
            )

    def set_constants(self):

        """ """
        self.x = np.linspace(0., self.length, self.nodes)
        self.J = self.I / self.area

    def get_dTq_dx(self, Tq, x):

        """ """
        T = Tq[0]
        q = Tq[1]
        self.set_TEproperties(T)
        dT_dx = (
            1. / self.k * (self.J * T * self.alpha - q)
            )
        self.set_ZT()
        dq_dx = (
```

```
            (self.rho * self.J ** 2. * (1. + self.ZT)) - self.J *
            self.alpha * q / self.k
            )
        dVs_dx = self.alpha * dT_dx
        dV_dx = self.alpha * dT_dx + self.rho * self.J
        dR_dx = self.rho / self.area

        return dT_dx, dq_dx, dVs_dx, dV_dx, dR_dx

    def solve_leg_once(self, q_h):

        """ """
        self.q_h = q_h
        self.y0 = np.array([self.T_h, self.q_h, 0, 0, 0])

        self.y = odeint(self.get_dTq_dx, y0=self.y0, t=self.x)

        self.T_x = self.y[:, 0]
        self.q_x = self.y[:, 1]
        self.Vs_x = self.y[:, 2]
        self.V_x = self.y[:, 3]
        self.R_int_x = self.y[:, 4]

        self.T_c = self.T_x[-1]
        self.q_c = self.q_x[-1]

        self.Vs = self.Vs_x[0] - self.Vs_x[-1]
        self.V = self.V_x[0] - self.V_x[-1]
        self.R_internal = self.R_int_x[-1]

        self.P_flux = self.J * self.V
        self.P = self.P_flux * self.area
        self.R_load = self.V / self.I

        self.eta = self.P / (self.q_h * self.area)

    def set_q_guess(self):

        """ """
        self.T_props = 0.5 * (self.T_h + self.T_c)
        self.set_TEproperties(T_props=self.T_props)
        delta_T = self.T_h - self.T_c
        self.q_c = - (
            self.alpha * self.T_c * self.J - delta_T / self.length *
        self.k - self.J ** 2 * self.length * self.rho
            )
        self.q_h = - (
            self.alpha * self.T_h * self.J - delta_T / self.length *
            self.k + self.J ** 2. * self.length * self.rho / 2.
            )
        self.q_c_guess = self.q_c
```

```
        self.q_h_guess = self.q_h
        self.q_guess = self.q_h

    def set_ZT(self):

        """ """
        self.ZT = self.alpha ** 2. * self.T_props / (self.k *
self.rho)

    def solve_leg(self):

        """ """
        self.T_h = self.T_h_conv
        self.T_c = self.T_c_conv

        self.fsolve_output = fsolve(self.get_error, x0=self.T_h -
1.)

    def get_error(self, T_h):

        """ """
        self.T_h = T_h[0]

        self.q_h_conv = self.U_hot * (self.T_h_conv - self.T_h)
        self.q_h = self.q_h_conv

        self.solve_leg_once(self.q_h)

        self.q_c_conv = self.U_cold * (self.T_c - self.T_c_conv)
        self.q_c_error = self.q_c - self.q_c_conv

        return self.q_c_error



        ==================== te_pair.py ====================

# Solution for a TE Pair

import numpy as np
import time
from scipy.optimize import fsolve

# User defined modules
import leg
reload(leg)

class TE_Pair(object):
    """ """

    def __init__(self):
```

```
        """ """
        self.R_desired = 2.22
        #self.leg_area_ratio = 0.7
        self.fill_fraction = 1.0
        self.length = 5.52e-3
        self.I = 0.0008
        self.Ptype = leg.Leg()
        self.Ntype = leg.Leg()
        self.Ptype.material = 'HMS'
        self.Ntype.material = 'MgSi'
        self.Ptype.area = (5.35 * 6.24e-6)
        self.Ntype.area = (5.65 * 6.19e-6)
        self.nodes = 10

        self.T_c_guess = 330.
        self.set_constants()

    def set_constants(self):

        """ """
        self.Ptype.length = self.length
        self.Ntype.length = self.length
        self.Ptype.nodes = self.nodes
        self.Ntype.nodes = self.nodes
        self.Ptype.I = self.I
        self.Ntype.I = - self.I
        self.Ptype.set_constants()
        self.Ntype.set_constants()

    def solve_te_pair_once(self):

        """ """
        #print "\nDid it get here then?"
        self.Ptype.solve_leg_once(self.Ptype.q_h)
        self.Ntype.solve_leg_once(self.Ntype.q_h)

    def set_q_guess(self):

        """ """
        self.Ptype.set_q_guess()
        self.Ntype.set_q_guess()
        #print "\n\n\nNtype q guess is ", self.Ptype.q_guess
        #print "Ntype q guess is ", self.Ntype.q_guess

    def get_error(self, knob_arr):

        """ """

        self.Ptype.q_h = knob_arr[0]
        self.Ntype.q_h = knob_arr[1]
```

```
        self.solve_te_pair_once()
        self.P_T_c_error = self.T_c - self.Ptype.T_c
        self.N_T_c_error = self.T_c - self.Ntype.T_c

        self.error = np.array([self.P_T_c_error,
self.N_T_c_error]).flatten()

        return self.error

    def solve_te_pair(self):

        """ """

        self.Ptype.T_h = self.T_h
        self.Ntype.T_h = self.T_h
        self.Ptype.T_c = self.T_c
        self.Ntype.T_c = self.T_c

        self.set_q_guess()
        self.set_constants()
        knob_arr0 = (
            np.array([self.Ptype.q_guess, self.Ntype.q_h_guess])
            )

        self.fsolve_output = fsolve(self.get_error, x0=knob_arr0)

        self.Vs = -self.Ntype.Vs + self.Ptype.Vs
        self.V = -self.Ntype.V + self.Ptype.V
        self.R_load = (
            self.Ntype.R_load + self.Ptype.R_load
            )
        self.R_internal = (
            self.Ntype.R_internal + self.Ptype.R_internal
            )

        self.R_ratio = self.R_load/self.R_internal
        self.P = (self.Ntype.P + self.Ptype.P) * 0.001
        # self.P_flux = self.P / self.area
        # self.eta = self.P / (self.q_h * self.area)

    def get_error1(self, I):

        """ """
        self.I = I[0]
        self.Ptype.T_h = self.T_h
        self.Ntype.T_h = self.T_h
        self.Ptype.T_c = self.T_c
        self.Ntype.T_c = self.T_c

        self.set_q_guess()
```

```
        self.set_constants()
        knob_arr0 = (
            np.array([self.Ptype.q_guess, self.Ntype.q_guess])
            )

        self.fsolve_output = fsolve(self.get_error, x0=knob_arr0)

        self.Vs = -self.Ntype.Vs + self.Ptype.Vs
        self.V = -self.Ntype.V + self.Ptype.V
        self.R_load = (
            self.Ntype.R_load + self.Ptype.R_load
            )
        self.R_internal = (
            self.Ntype.R_internal + self.Ptype.R_internal
            )

        self.P = (self.Ntype.P + self.Ptype.P) * 0.001
        self.R_ratio = self.R_desired/self.R_internal
        self.R_error = self.R_desired - self.R_load

        return self.R_error

    def solve_te_pair_R(self):
        """ """
        self.fsolve_output1 = fsolve(self.get_error1, x0=self.I)

        ==================== mat_prop.py ====================

"""Module containing set properties function."""

import numpy as np

def import_raw_property_data(self):

    """Imports and sets values for material properties as a function
    of temperature."""

    print "running import_raw_property_data"

    if self.material == "HMS":
        # Raw data taken from Luo et al. HMS is p-type
        poly_deg = 3
        # print "Curve fitting for HMS"
        self.alpha_raw = np.array([[830.9667,232.42625],
                                   [825.855,231.47188],
                                   [810.8334,230.17526],
                                   [795.8501,235.14671],
                                   [780.9,233.48876],
                                   [765.8434,234.53112],
                                   [750.8484,231.80171],
                                   [735.87,231.16316],
```

```
                                [720.9167,232.57991],
                                [705.88,238.2938],
                                [690.8833,238.88064],
                                [675.8517,228.6744],
                                [660.8784,227.07233],
                                [645.8851,228.60436],
                                [630.9001,228.28546],
                                [615.83,223.78528],
                                [600.8667,219.25532],
                                [585.84,218.75806],
                                [570.845,211.26455],
                                [555.85,208.26649],
                                [540.8167,207.52015],
                                [525.96,202.52498],
                                [510.85,194.35518],
                                [495.905,184.67011],
                                [480.85,185.92441],
                                [465.88,186.66846],
                                [450.85,175.34614],
                                [435.9083,171.48397],
                                [420.8333,162.04891],
                                [405.8367,153.29339],
                                [390.8833,152.25655],
                                [375.845,143.86649],
                                [360.78333,139.59344],
                                [345.96167,131.00214],
                                [330.9,128.76147],
                                [315.78833,117.81014]])


        self.alpha_params = np.polyfit(
            self.alpha_raw[:, 0], self.alpha_raw[:, 1], poly_deg
            )

        self.k_raw = np.array([[322.6, 2.5579899317],
                               [372.9, 2.51810604],
                               [423.3, 2.5371132561],
                               [473.3, 2.5417150908],
                               [523.3, 2.578562122],
                               [573.3, 2.5915271088],
                               [623.3, 2.5654607341],
                               [673.3, 2.5650773755],
                               [723.3, 2.5735915738],
                               [773.3, 2.6374224558],
                               [823.3, 2.7425225851],
                               [873.3, 2.9102448442]])
        self.k_params = np.polyfit(
            self.k_raw[:, 0], self.k_raw[:, 1], poly_deg
            )

        self.sigma_raw = np.array([[315.705, 447.48359],
```

```
                                [330.76667, 427.31343],
                                [345.84167, 409.70235],
                                [360.81667, 392.78365],
                                [375.785, 378.55348],
                                [390.8333, 365.56717],
                                [405.84, 354.14173],
                                [420.8833, 343.39514],
                                [435.8567, 335.56024],
                                [450.85, 324.76268],
                                [465.8633, 316.43513],
                                [480.8667, 306.77187],
                                [495.8534, 301.23106],
                                [510.85, 294.17404],
                                [525.8617, 290.47311],
                                [540.85, 285.56701],
                                [555.885, 280.54875],
                                [570.895, 274.45275],
                                [585.8233, 271.61472],
                                [600.8817, 268.3056],
                                [615.875, 271.1431],
                                [630.85, 267.11241],
                                [645.8533, 265.20722],
                                [660.895, 264.45594],
                                [675.8417, 264.3013],
                                [690.8834, 270.92926],
                                [705.88, 263.90337],
                                [720.9167, 262.34766],
                                [735.9333, 264.72492],
                                [750.8651, 264.79832],
                                [765.8734, 278.1682],
                                [780.8667, 274.54929],
                                [795.8667, 282.64445],
                                [810.9, 284.56422],
                                [825.905, 287.06549],
                                [831.0483, 293.59086]])
        self.sigma_params = np.polyfit(
            self.sigma_raw[:, 0], self.sigma_raw[:, 1], poly_deg
            )

        self.density_raw = np.array([[322.6, 4786.],
                                     [372.9, 4786.],
                                     [423.3, 4786.],
                                     [473.3, 4786.],
                                     [523.3, 4786.],
                                     [573.3, 4786.],
                                     [623.3, 4786.],
                                     [673.3, 4786.],
                                     [723.3, 4786.],
                                     [773.3, 4786.],
                                     [823.3, 4786.],
                                     [873.3, 4786.]])
```

```
        self.density_params = np.polyfit(
            self.density_raw[:, 0], self.density_raw[:, 1], poly_deg
            )


        self.c_raw = np.array([[322.6, 582.85],
                               [372.9,  592.5],
                               [423.3,  604.46],
                               [473.3,  615.38],
                               [523.3,  631.62],
                               [573.3,  644.62],
                               [623.3,  652.11],
                               [673.3,  667.44],
                               [723.3,  678.1],
                               [773.3,  689.7],
                               [823.3,  696.27],
                               [873.3,  703.79]])
        self.c_params = np.polyfit(
            self.c_raw[:, 0], self.c_raw[:, 1], poly_deg
            )

    if self.material == "MgSi":
        # Raw data comes from Gao et al. MgSi is n-type
        poly_deg = 2

        self.alpha_raw = np.array([[304.97667, -76.77944],
                                   [320.13334, -78.54224],
                                   [335.07833, -90.72432],
                                   [350.11667, -90.27498],
                                   [365.15, -95.12417],
                                   [380.1783, -94.27527],
                                   [395.085, -102.2901],
                                   [410.1, -99.26342],
                                   [425.1283, -106.00969],
                                   [440.1167, -105.58557],
                                   [455.1167, -107.40208],
                                   [470.1333, -110.31651],
                                   [485.185, -114.40619],
                                   [500.1, -123.45917],
                                   [515.095, -121.51015],
                                   [530.0334, -126.49965],
                                   [545.0667, -132.28749],
                                   [560.1333, -128.94191],
                                   [575.1034, -133.0895],
                                   [590.1, -137.9517],
                                   [605.1933, -143.75023],
                                   [620.15, -147.52096],
                                   [635.1684, -157.04423],
                                   [650.0533, -149.56359],
                                   [665.0683, -150.01531],
                                   [680.1833, -158.23988],
```

```
                            [695.1117, -174.23052],
                            [710.0667, -175.58246],
                            [725.0367, -170.67081],
                            [740.1167, -167.63654],
                            [755.12, -186.00031],
                            [770.0717, -185.53065],
                            [785.0617, -181.88676],
                            [800.05, -197.40704],
                            [815.1167, -200.92754]])
self.alpha_params = np.polyfit(
    self.alpha_raw[:, 0], self.alpha_raw[:, 1], poly_deg
    )



self.k_raw = np.array([[323.15, 3.13731],
                       [373.15, 3.00759],
                       [423.15, 2.91603],
                       [473.15, 2.84598],
                       [523.15, 2.83228],
                       [573.15, 2.77579],
                       [623.15, 2.75821],
                       [673.15, 2.6791],
                       [723.15, 2.61775],
                       [773.15, 2.5552],
                       [803.15, 2.51657]])
self.k_params = np.polyfit(
    self.k_raw[:, 0], self.k_raw[:, 1], poly_deg
    )

self.sigma_raw = np.array([[304.97667,2025.4303064],
                           [320.13334,1953.0404522],
                           [335.07833,1929.119928],
                           [350.11667,1893.2014767],
                           [365.15,1813.2854751],
                           [380.1783,1849.1840193],
                           [395.085,1674.9071539],
                           [410.1,1731.5919452],
                           [425.1283,1657.625887],
                           [440.1167,1596.9892047],
                           [455.1167,1575.9967393],
                           [470.1333,1476.6277729],
                           [485.185,1417.9797169],
                           [500.1,1398.2341828],
                           [515.095,1374.0772588],
                           [530.0334,1332.5900323],
                           [545.0667,1319.6739576],
                           [560.1333,1285.4625348],
                           [575.1034,1288.3484849],
                           [590.1,1207.2220289],
                           [605.1933,1194.4769917],
```

```
                                    [620.15,1132.0255237],
                                    [635.1684,1135.2257844],
                                    [650.0533,1223.7744333],
                                    [665.0683,1109.2040017],
                                    [680.1833,1047.5815571],
                                    [695.1117,1056.8158787],
                                    [710.0667,1008.1394804],
                                    [725.0367,936.9948972],
                                    [740.1167,952.0983141],
                                    [755.12,856.7463382],
                                    [770.0717,904.9065428],
                                    [785.0617,847.1685289],
                                    [800.05,764.5969332],
                                    [815.1167,743.0947173]])
        self.sigma_params = np.polyfit(self.sigma_raw[:, 0],
self.sigma_raw[:, 1],
                               poly_deg)

        self.density_raw = np.array([[283.888641142, 2900.],
                                     [396.056571319, 2900.],
                                     [573.510861948, 2900.],
                                     [786.035548194, 2900.],
                                     [856.520354208, 2900.],
                                     [901.20405173, 2900.]])
        self.density_params = np.polyfit(
            self.density_raw[:, 0], self.density_raw[:, 1], poly_deg
            )


        self.c_raw = np.array([[323.15, 1905.],
                               [373.15, 1823.],
                               [423.15, 1752.],
                               [473.15, 1698.],
                               [523.15, 1644.],
                               [573.15, 1587.],
                               [623.15, 1526.],
                               [673.15, 1468.],
                               [723.15, 1423.],
                               [773.15, 1389.],
                               [803.15, 1368.]])
        self.c_params = np.polyfit(
            self.c_raw[:, 0], self.c_raw[:, 1], poly_deg
            )

def set_properties_v_temp(self, T_props):

    """ Sets properties based on polynomial fit values.
    """
    try:
        self.alpha_params
    except AttributeError:
```

```
        self.import_raw_property_data()

    # Seebeck coefficient (V/K)
    self.alpha = 0.7 * (np.polyval(self.alpha_params, T_props) *
1.e-6)

    # thermal conductivity (W/m-K)
    self.k = np.polyval(self.k_params, T_props)

    # electrical conductivity (S/m)
    self.sigma = np.polyval(self.sigma_params, T_props) * 1.e2

    # electrical resistivity (Ohm-m)
    self.rho = 1. / self.sigma

    # density (kg/m3)
    self.density = np.polyval(self.density_params, T_props)

    # specific heat (J/kg-K)
    self.c = np.polyval(self.c_params, T_props)

    # effective thermal mass, density times c (J/m3-K)
    self.C = self.density * self.c

def set_TEproperties(self, T_props):

    """Sets TE properties

    Inputs:

    T_props : temperature (K) at which properties are to be
evaluated

    This method exists to separater materials with constant
properties
    from materials with temperature dependent properties.  It uses
    set_properties_v_temp for the latter type of materials.

    """

    self.T_props = T_props

    # Materials with tabulated properties
    if self.material == 'HMS':
        self.set_properties_v_temp(T_props)

    elif self.material == 'MgSi':
        self.set_properties_v_temp(T_props)


        ==================== fin_inst.py ====================
```

```
# produces bunch of results as needed using straight fins

# Distribution Modules
import matplotlib.pyplot as plt
import os,sys
import numpy as np

# User Defined Modules
cmd_folder = os.path.dirname(os.path.abspath('../Modules/hx.py'))
if cmd_folder not in sys.path:
    sys.path.insert(0, cmd_folder)
import hx
reload(hx)

area_ratio = 0.7
fill_fraction = 3.10e-2
leg_length = 3.56e-4
#R_desired = 2.5
current = 10.0

hx_fins0 = hx.HX()

hx_fins0.width = 0.55
hx_fins0.exh.height = 3.5e-2
hx_fins0.length = 0.55
hx_fins0.te_pair.length = leg_length
hx_fins0.te_pair.leg_area_ratio = area_ratio
hx_fins0.te_pair.fill_fraction = fill_fraction

hx_fins0.te_pair.set_leg_areas()

hx_fins0.te_pair.Ntype.material = 'MgSi'
hx_fins0.te_pair.Ptype.material = 'HMS'

hx_fins0.type = 'counter'

hx_fins0.exh.enh = hx_fins0.exh.set_enhancement('IdealFin')
hx_fins0.exh.enh.thickness = 1.e-3
hx_fins0.exh.enh.spacing = 1.26e-3

hx_fins0.cool.enh = hx_fins0.cool.set_enhancement('IdealFin')
#hx_fins0.cool.enh = hx_fins0.cool.set_enhancement('CircularDuct')
hx_fins0.cool.enh.thickness = 1.e-3
hx_fins0.cool.enh.spacing = 1.e-3

hx_fins0.exh.T_inlet = 800.
hx_fins0.cool.T_inlet_set = 300.
hx_fins0.cool.T_outlet = 310.

hx_fins0.set_mdot_charge()
```

```
# print "mdot is ", hx_fins0.exh.mdot
# # ============================================
# # this part runs fin inst once
#hx_fins0.R_desired = R_desired
#hx_fins0.solve_hx_R()
# print "Solved HX once ... "
# # ============================================



# ============================================

hx_fins0.te_pair.I = current
hx_fins0.solve_hx()
print "Solved HX once ... "
# this part runs optimization once
print "Running optimization ... "
#hx_fins0.optimize_I()

print "power net:", hx_fins0.power_net * 1000., 'W'
print "power raw:", hx_fins0.te_pair.power_total * 1000., 'W'
print "pumping power:", hx_fins0.Wdot_pumping * 1000., 'W'
hx_fins0.exh.volume = hx_fins0.exh.height * hx_fins0.exh.width *
hx_fins0.length
print "exhaust volume:", hx_fins0.exh.volume * 1000., 'L'
print "exhaust power density:", hx_fins0.power_net /
hx_fins0.exh.volume, 'kW/m^3'

# ============================================



# # ============================================
# # gridcheck for HX using straight fins

# hx_fins0.solve_hx_R()

# SIZE = 15
# nodes = np.arange(20, 150, SIZE)
# power_net = np.zeros(nodes.size)

# for i in range(nodes.size):
#     hx_fins0.nodes = nodes[i]
#     hx_fins0.solve_hx_R()
#     print hx_fins0.nodes, " nodes"
#     nodes[i] = hx_fins0.nodes
#     power_net[i] = hx_fins0.power_net * 1000

# print "\nPlotting..."

# # Plot configuration
```

```
# FONTSIZE = 10
# plt.rcParams['axes.labelsize'] = FONTSIZE
# plt.rcParams['axes.titlesize'] = FONTSIZE
# plt.rcParams['legend.fontsize'] = FONTSIZE
# plt.rcParams['xtick.labelsize'] = FONTSIZE
# plt.rcParams['ytick.labelsize'] = FONTSIZE
# plt.rcParams['lines.linewidth'] = 1.5

# plt.close()

# plt.figure()
# plt.plot(nodes, power_net)

# plt.xlim(nodes.min() - 1., nodes.max() + 1.)
# plt.ylim(power_net.min() - 500., power_net.max() + 100.)
# plt.xlabel('Number of nodes')
# plt.ylabel('HX net power (W)')
# plt.grid()
# plt.show()
# # ==========================================

# # # ==========================================
# # # # Haiyan Fateh
# # # # single HX run for a given set of inputs
# # # # Plot temperatures and other stuff

# # loading optimized results
# data_dir = '../Output/'

# length = np.load(data_dir + 'length.npy')
# fill_fraction = np.load(data_dir + 'fill_fraction.npy')
# leg_area_ratio = np.load(data_dir + 'leg_area_ratio.npy')
# R_desired = np.load(data_dir + 'R_desired.npy')
# #R_internal = np.load(data_dir + 'R_internal.npy')
# #leg_pairs = np.load(data_dir + 'leg_pairs.npy')
# #power_net = np.load(data_dir + 'power_net.npy')
# #current = np.load(data_dir + 'current.npy')
# #V = np.load(data_dir + 'V.npy')
# #Vs = np.load(data_dir + 'Vs.npy')

# SIZE = length.size

# T_h_conv = np.zeros([SIZE, hx_fins0.nodes])
# T_h_TE = np.zeros([SIZE, hx_fins0.nodes])
# T_c_conv = np.zeros([SIZE, hx_fins0.nodes])
# T_c_TE = np.zeros([SIZE, hx_fins0.nodes])
# power_net = np.zeros(SIZE)

# # solve all the optimized cases
# #for i in range(SIZE):
# for i in range(1):
```

```
#     hx_fins0.te_pair.length = length[i]
#     hx_fins0.te_pair.fill_fraction = fill_fraction[i]
#     hx_fins0.R_desired = R_desired[i]
#     hx_fins0.leg_area_ratio = leg_area_ratio[i]
#     hx_fins0.solve_hx_R()

#     # store results for plotting
#     power_net[i] = hx_fins0.power_net
#     T_h_conv[i,:] = hx_fins0.exh.T_nodes
#     T_h_TE[i,:] = hx_fins0.te_pair.T_h_nodes
#     T_c_conv[i,:] = hx_fins0.cool.T_nodes
#     T_c_TE[i,:] = hx_fins0.te_pair.T_c_nodes

# # plotting
# print "\nPlotting..."

# # Plot configuration
# FONTSIZE = 14
# plt.rcParams['axes.labelsize'] = FONTSIZE
# plt.rcParams['axes.titlesize'] = FONTSIZE
# plt.rcParams['legend.fontsize'] = FONTSIZE
# plt.rcParams['xtick.labelsize'] = FONTSIZE
# plt.rcParams['ytick.labelsize'] = FONTSIZE
# plt.rcParams['lines.linewidth'] = 1.5

# plt.close()
# plt.figure()
# fig1 = plt.figure()

# for i in range(SIZE):
#     plt.plot(hx_fins0.x * 100., T_h_conv[i,:])
#     plt.plot(hx_fins0.x * 100., T_h_TE[i,:])
#     plt.plot(hx_fins0.x * 100., T_c_conv[i,:])
#     plt.plot(hx_fins0.x * 100., T_c_TE[i,:])

# plt.grid()
# plt.xlabel('Distance Along HX (cm)')
# plt.ylabel('Temperature (K)')
# plt.show()
# # # ========================================

# # # ========================================
# # # # Haiyan Fateh
# # # # plot
# # # # Optimization of TE design space with straight fins
# # # # vary input R_desired and optimize

# SIZE = 10

# length = np.zeros(SIZE)
# fill_fraction = np.zeros(SIZE)
```

```
# leg_area_ratio = np.zeros(SIZE)
# R_desired = np.zeros(SIZE)
# R_internal = np.zeros(SIZE)
# leg_pairs = np.zeros(SIZE)
# power_net = np.zeros(SIZE)
# current = np.zeros(SIZE)
# V = np.zeros(SIZE)
# Vs = np.zeros(SIZE)

# hx_fins0.I = 10.
# hx_fins0.te_pair.length = 0.00034001
# hx_fins0.te_pair.fill_fraction = 0.042055
# hx_fins0.te_pair.leg_area_ratio = 0.69786
# R_desired = np.linspace(4., 10., SIZE)
# hx_fins0.solve_hx_R()

# print "\n"
# print "\nDone solving ... "
# print "\n"
# print "Running optimization now ... "

# for i in range(SIZE):
#     hx_fins0.R_desired = R_desired[i]
#     hx_fins0.optimize()
#     length[i] = hx_fins0.te_pair.length
#     fill_fraction[i] = hx_fins0.te_pair.fill_fraction
#     leg_area_ratio[i] = hx_fins0.te_pair.leg_area_ratio
#     R_desired[i] = hx_fins0.R_desired
#     leg_pairs[i] = hx_fins0.leg_pairs
#     power_net[i] = hx_fins0.power_net
#     current[i] = hx_fins0.te_pair.I
#     R_internal[i] = hx_fins0.te_pair.R_internal_nodes_total
#     V[i] = hx_fins0.te_pair.V_nodes_total
#     Vs[i] = hx_fins0.te_pair.Vs_nodes_total

# data_dir = '../Output/fin_opt/'
# np.save(data_dir + 'length', length)
# np.save(data_dir + 'fill_fraction', fill_fraction)
# np.save(data_dir + 'leg_area_ratio', leg_area_ratio)
# np.save(data_dir + 'R_desired', R_desired)
# np.save(data_dir + 'R_internal', R_internal)
# np.save(data_dir + 'leg_pairs', leg_pairs)
# np.save(data_dir + 'power_net', power_net)
# np.save(data_dir + 'current', current)
# np.save(data_dir + 'V', V)
# np.save(data_dir + 'Vs', Vs)

# # ========================================

# # ========================================
# # # Haiyan plot
```

```
# # # Power dependence on total load resistance
# SIZE = 3
# R_desired = np.linspace(0.5, 5.0, SIZE)
# Power = np.zeros(SIZE)
# R_load = np.zeros(SIZE)
# R_internal = np.zeros(SIZE)
# current = np.zeros(SIZE)
# Vs = np.zeros(SIZE)
# V = np.zeros(SIZE)

# for i in range(SIZE):
#     hx_fins0.R_desired = R_desired[i]
#     hx_fins0.solve_hx_R()
#     Power[i] = hx_fins0.te_pair.power_total
#     R_load[i] = hx_fins0.te_pair.R_load_nodes_total
#     R_internal[i] = hx_fins0.te_pair.R_internal_nodes_total
#     current[i] = hx_fins0.te_pair.I
#     Vs[i] = hx_fins0.te_pair.Vs_nodes_total
#     V[i] = hx_fins0.te_pair.V_nodes_total

# print "\nProgram finished."
# print "\nPlotting..."

# # Plot configuration
# FONTSIZE = 20
# plt.rcParams['axes.labelsize'] = FONTSIZE
# plt.rcParams['axes.titlesize'] = FONTSIZE
# plt.rcParams['legend.fontsize'] = FONTSIZE
# plt.rcParams['xtick.labelsize'] = FONTSIZE
# plt.rcParams['ytick.labelsize'] = FONTSIZE
# plt.rcParams['lines.linewidth'] = 1.5

# plt.figure()
# plt.plot(R_desired, Power)

# plt.xlabel('Total HX load resistance (ohm)')
# plt.ylabel('Power output (kW)')
# plt.grid()
# plt.show()
# # ========================================


   ==================== te_volume_vs_fill.py ====================

# Distribution Modules
import numpy as np
import matplotlib.pyplot as plt
import os
import sys
import time
```

```
# User Defined Modules
cmd_folder = os.path.dirname(os.path.abspath('../Modules/hx.py'))
if cmd_folder not in sys.path:
    sys.path.insert(0, cmd_folder)

import te_pair
reload(te_pair)

# instantiate a te_design object
te_design = te_pair.TE_Pair()

te_design.nodes = 10

te_design.Ntype.material = 'MgSi'
te_design.Ptype.material = 'HMS'
te_design.Ptype.area = (3.e-3) ** 2

te_design.T_c_conv = 300.  # cold side convection temperature (K)
te_design.T_h_conv = 680.  # hot side convection temperature (K)

te_design.U_cold = 8.
# cold side overall heat transfer coeffcient (kW / (m ** 2 * K))
te_design.U_hot = 2.
# hot side overall heat transfer coeffcient (kW / (m ** 2 * K))

#data_dir = '../Output/Standalone_TE/Sets/'
#data_dir = '../Output/Standalone_TE/Sets1/'
data_dir = '../Output/Standalone_TE/Sets4/'
length = np.load(data_dir + 'length.npy')
fill_fraction = np.load(data_dir + 'fill_fraction.npy')
leg_area_ratio = np.load(data_dir + 'leg_area_ratio.npy')
R_ratio = np.load(data_dir + 'R_ratio.npy')
R_desired = np.load(data_dir + 'R_desired.npy')
Power = np.load(data_dir + 'Power.npy')
Power_flux = np.load(data_dir + 'Power_flux.npy')
current = np.load(data_dir + 'current.npy')
T_h = np.load(data_dir + 'T_h.npy')
T_c = np.load(data_dir + 'T_c.npy')

area_total = 1.0  # m^2
area_pair = np.zeros([5, length.size/5])
leg_pairs = np.zeros([5, length.size/5])
leg_occupied_area = np.zeros([5, length.size/5])
leg_length  = np.zeros([5, length.size/5])
te_volume = np.zeros([5, length.size/5])

for i in range(5):

    for j in range(length.size/5):

        te_volume[i, j] = (
```

```
            1 * fill_fraction[i, j] * length[i, j]
            )

print "\nPlotting..."

save_dir = "../Results/"
# Plot configuration
FONTSIZE = 14
plt.rcParams['axes.labelsize'] = FONTSIZE
plt.rcParams['axes.titlesize'] = FONTSIZE
plt.rcParams['legend.fontsize'] = FONTSIZE
plt.rcParams['xtick.labelsize'] = FONTSIZE
plt.rcParams['ytick.labelsize'] = FONTSIZE
plt.rcParams['lines.linewidth'] = 1.5

plt.close()

text = '''Optimal power flux: '''
# text = ''' Optimal power flux
# ($\Delta$T = 150 K) = 1.44 kW/m$^3$
# ($\Delta$T = 200 K) = 2.52 kW/m$^3$
# ($\Delta$T = 250 K) = 3.87 kW/m$^3$
# ($\Delta$T = 300 K) = 5.47 kW/m$^3$
# ($\Delta$T = 350 K) = 7.31 kW/m$^3$'''

# label = (
#     np.array(['($\Delta$T = 150 K) = 1.44 kW/m$^3$', '($\Delta$T =
200 K) = 2.52 kW/m$^3$', '($\Delta$T = 250 K) = 3.87 kW/m$^3$',
'($\Delta$T = 300 K) = 5.47 kW/m$^3$', '($\Delta$T = 350 K) = 7.31
kW/m$^3$'])
#     )

label = (
    np.array(['($\Delta$T = 300 K) = 5.47 kW/m$^3$', '($\Delta$T =
350 K) = 7.31 kW/m$^3$', '($\Delta$T = 400 K) = 9.38 kW/m$^3$',
'($\Delta$T = 450 K) = 11.65 kW/m$^3$', '($\Delta$T = 500 K) = 14.11
kW/m$^3$'])
    )

ax1 = plt.subplot(111)

#plt.figure()
plt.figtext(0.14, 0.65, text, size=14)

for i in range(5):
    plt.plot(fill_fraction[i,:] * 1.e2, te_volume[i,:] * 1.e6,
label=label[i])

plt.legend(loc="center left", prop={'size':12})
plt.xlabel('Fill fraction (%)')
#plt.ylabel('Power (mW)')
```

```
plt.ylabel('Volume of TE material (cm$^3$)')

ax2 = plt.twinx()
plt.ylabel('Leg length (mm)')

plt.grid()
plt.show()
plt.savefig(save_dir + "te_volume_vs_fill1.pdf")
```

```
 ==================== te_varied_R_ratio_plot.py ====================

# Distribution Modules
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import axes3d
import os
import sys

# User Defined Modules
cmd_folder = os.path.dirname(os.path.abspath('../Modules/hx.py'))
if cmd_folder not in sys.path:
    sys.path.insert(0, cmd_folder)

data_dir = '../Results/Resistance_data/'

P = np.load(data_dir + 'P.npy')
P_flux = np.load(data_dir + 'P_flux.npy')
R_ratio = np.load(data_dir + 'R_ratio.npy')
R_internal = np.load(data_dir + 'R_internal.npy')
T_hot_TE = np.load(data_dir + 'T_hot_TE.npy')
T_cold_TE = np.load(data_dir + 'T_cold_TE.npy')

# # label = (
# #     np.array(['$\Delta$T = 350 K', '$\Delta$T = 300 K',
'$\Delta$T = 250 K', '$\Delta$T = 200 K', '$\Delta$T = 150 K'])
# #     )

# label = (
#     np.array(['$\Delta$T = 500 K', '$\Delta$T = 450 K', '$\Delta$T
= 400 K', '$\Delta$T = 350 K', '$\Delta$T = 300 K'])
#     )

# # print "\nPlotting..."

# # # # For saving plot
# # save_dir = "../Results/"

# # # Plot configuration
# # FONTSIZE = 14
# # plt.rcParams['axes.labelsize'] = FONTSIZE
# # plt.rcParams['axes.titlesize'] = FONTSIZE
```

```
# # plt.rcParams['legend.fontsize'] = FONTSIZE
# # plt.rcParams['xtick.labelsize'] = FONTSIZE
# # plt.rcParams['ytick.labelsize'] = FONTSIZE
# # plt.rcParams['lines.linewidth'] = 1.5
# # plt.rcParams['lines.markersize'] = 10

# # plt.close()
# # plt.figure()

# # for i in range(5):
# #     plt.plot(R_ratio[i,:], P_flux[i, :], label = label[i])

# # plt.grid()
# # plt.xlabel('Load resistance / Internal resistance')
# # plt.ylabel('Power Flux (kW/m$^2$)')
# # plt.legend(loc="upper right")

# # plt.savefig(save_dir + "P_vs_R_ratio_TE.pdf")
# # plt.show()


# # # Analytical plots

# # delta_T = np.array([150., 200., 250., 300., 350.])
# # alpha_n_lower = np.array([-76., -76., -76., -76., -76.])
# # alpha_n_higher = np.array([-106.33, -123.45, -130.66, -141.0, -
149.0])
# # alpha_n_average = (alpha_n_lower + alpha_n_higher) * 1.e-6 / 2

# # alpha_p_lower = np.array([233.0, 233.0, 233.0, 233.0, 233.0])
# # alpha_p_higher = np.array([238.0, 227, 223.3, 208.0, 190.66])
# # alpha_p_average = (alpha_p_lower + alpha_p_higher) * 1.e-6 / 2

# delta_T = np.array([500., 450., 400., 350., 300.])
# alpha_n_lower = np.array([-76., -76., -76., -76., -76.])
# alpha_n_higher = np.array([-197.0, -179.0, -174.0, -149.0, -
141.0])
# alpha_n_average = (alpha_n_lower + alpha_n_higher) * 1.e-6 / 2

# alpha_p_lower = np.array([233.0, 233.0, 233.0, 233.0, 233.0])
# alpha_p_higher = np.array([136.0, 153.0, 178.0, 190.0, 208.0])
# alpha_p_average = (alpha_p_lower + alpha_p_higher) * 1.e-6 / 2

# P_analytical = np.zeros([5, 50])

# for i in range(5):
#     for j in range(50):
#         P_analytical[i, j] = (
#             (((alpha_p_average[i] - alpha_n_average[i]) *
(T_hot_TE[i,j] - T_cold_TE[i, j])) ** 2 * R_ratio[i, j]) /
#             ((1 + R_ratio[i, j]) ** 2 * R_internal[i, j])
```

```
#             )

# area = (P[0,0] / P_flux[0,0])
# P_analytical_flux = P_analytical/area

# print "\nPlotting..."

# # # For saving plot
# save_dir = "../Results/"

# # Plot configuration
# FONTSIZE = 14
# plt.rcParams['axes.labelsize'] = FONTSIZE
# plt.rcParams['axes.titlesize'] = FONTSIZE
# plt.rcParams['legend.fontsize'] = FONTSIZE
# plt.rcParams['xtick.labelsize'] = FONTSIZE
# plt.rcParams['ytick.labelsize'] = FONTSIZE
# plt.rcParams['lines.linewidth'] = 1.5
# plt.rcParams['lines.markersize'] = 10

# plt.close()
# plt.figure()

# for i in range(5):
#     plt.plot(R_ratio[i,:], P_analytical_flux[i, :], linestyle='-
.', label=label[i])
#     plt.plot(R_ratio[i,:], P_flux[i, :] * 1.e3, label = label[i])

# text1 = "- Analytical "
# text2 = "... Numerical "

# font = {'weight':'bold'}
# plt.text(4., 0.85 * P_analytical_flux.max(), text1, **font)
# plt.text(4., 0.90 * P_analytical_flux.max(), text2, **font)
# plt.grid()
# plt.ylim(0, P_analytical_flux.max() * 1.1)
# plt.xlabel('Load resistance / Internal resistance')
# plt.ylabel('Power Flux (kW/m$^2$)')
# plt.legend(loc="upper right", prop={'size':12})

# plt.savefig(save_dir +
"P_vs_R_ratio_TE_both_theoretical_experimental.pdf")
# plt.show()
```

`==================== te_design_variable_sets.py ====================`

```
# # Produces various combinations of optimized parameters
# # Input R_desired is changed in every run
# generates data for 3d plot for te design space optimization using
a
# standalone TE pair
```

```
# Distribution Modules
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
import os
import sys
import time

# User Defined Modules
cmd_folder = os.path.dirname(os.path.abspath('../Modules/hx.py'))
if cmd_folder not in sys.path:
    sys.path.insert(0, cmd_folder)

import te_pair
reload(te_pair)

# instantiate a te_design object
te_design = te_pair.TE_Pair()

te_design.nodes = 10

# declare materials to be used for property calculations
te_design.Ntype.material = 'MgSi'
te_design.Ptype.material = 'HMS'

te_design.pairs = 1.
te_design.Ptype.area = (3.e-3) ** 2
#te_design.area = (5.e-3) ** 2

# T_hot_conv = np.array([450., 500., 550., 600., 650.])
# T_cold_conv = np.array([300., 300., 300., 300., 300.])

T_hot_conv = np.array([600., 650., 700., 750., 800.])
T_cold_conv = np.array([300., 300., 300., 300., 300.])

# te_design.T_c_conv = 300.  # cold side convection temperature (K)
# te_design.T_h_conv = 680.  # hot side convection temperature (K)

te_design.U_cold = 8.
# cold side overall heat transfer coeffcient (kW / (m ** 2 * K))
te_design.U_hot = 2.
# hot side overall heat transfer coeffcient (kW / (m ** 2 * K))

# # Produces various combinations of optimized parameters
# # Input R_desired is changed in every run
SIZE = 10
# # optimization based on length
# length_guess = np.linspace(0.1e-3, 1.0e-3, SIZE)
# fill_fraction_guess = 0.3
# R_desired_guess = 0.002
```

```
# optimization based on length
length_guess = 0.0005
fill_fraction_guess = np.linspace(0.1, 0.5, SIZE)
R_desired_guess = 0.003

length = np.zeros([5, SIZE])
fill_fraction = np.zeros([5, SIZE])
leg_area_ratio = np.zeros([5, SIZE])
R_desired = np.zeros([5, SIZE])
Power = np.zeros([5, SIZE])
Power_flux = np.zeros([5, SIZE])
R_ratio = np.zeros([5, SIZE])
area_ratio = np.zeros([5, SIZE])
current = np.zeros([5, SIZE])
T_h = np.zeros([5, SIZE])
T_c = np.zeros([5, SIZE])

for i in range(5):
    print "i = ", i

    te_design.T_h_conv = T_hot_conv[i]
    te_design.T_c_conv = T_cold_conv[i]

    for j in range(SIZE):

        #e_design.length = length_guess[i]
        te_design.length = length_guess
        te_design.fill_fraction = fill_fraction_guess[j]
        te_design.R_desired = R_desired_guess

        te_design.set_constants()
        te_design.solve_te_pair_R()
        te_design.optimize()

        length[i, j] = te_design.length
        fill_fraction[i, j] = te_design.fill_fraction
        leg_area_ratio[i, j] = te_design.leg_area_ratio
        R_desired[i, j] = te_design.R_desired
        R_ratio[i, j] = te_design.R_ratio
        Power[i, j] = te_design.P
        Power_flux[i, j] = te_design.P_flux
        area_ratio[i, j] = te_design.leg_area_ratio
        current[i, j] = te_design.I
        T_h[i, j] = te_design.T_h
        T_c[i, j] = te_design.T_c

data_dir = '../Output/Standalone_TE/Sets4/'
np.save(data_dir + 'length', length)
np.save(data_dir + 'fill_fraction', fill_fraction)
np.save(data_dir + 'leg_area_ratio', leg_area_ratio)
```

```
np.save(data_dir + 'R_ratio', R_ratio)
np.save(data_dir + 'R_desired', R_desired)
np.save(data_dir + 'Power', Power)
np.save(data_dir + 'Power_flux', Power_flux)
np.save(data_dir + 'current', current)
np.save(data_dir + 'T_h', T_h)
np.save(data_dir + 'T_c', T_c)

#execfile('plot_te_design_space_3dscatter.py')
```

=================== te_design_optimize.py ===================

```
# Runs number of solutions and saves data regarding optimization
# The data produced here is saved for plotting later

# Distribution Modules
import numpy as np
import os
import sys
import time

# User Defined Modules
cmd_folder = os.path.dirname(os.path.abspath('../Modules/hx.py'))
if cmd_folder not in sys.path:
    sys.path.insert(0, cmd_folder)

import te_pair
reload(te_pair)

# instantiate a te_design object
te_design = te_pair.TE_Pair()

te_design.nodes = 10

te_design.Ntype.material = 'MgSi'
te_design.Ptype.material = 'HMS'
te_design.Ptype.area = (3.e-3) ** 2

fill_fraction = 0.3
length = 0.0005
area_ratio = 0.7004
R_desired = 0.00593129

te_design.fill_fraction = fill_fraction
te_design.length = length
te_design.leg_area_ratio = area_ratio
te_design.R_desired = R_desired

te_design.T_c_conv = 300.  # cold side convection temperature (K)
```

```
#te_design.T_h_conv = 680.  # hot side convection temperature (K)
te_design.T_h_conv = 800.  # hot side convection temperature (K)

te_design.U_cold = 8.
# cold side overall heat transfer coeffcient (kW / (m ** 2 * K))
te_design.U_hot = 2.
# hot side overall heat transfer coeffcient (kW / (m ** 2 * K))

te_design.set_constants()
te_design.solve_te_pair_R()

te_design.optimize()

#=================================
#postprocessing after optimization
#=================================

R_internal = te_design.R_internal
R_desired = te_design.R_desired

leg_area_ratio = te_design.leg_area_ratio
fill_fraction = te_design.fill_fraction
leg_area_ratio = te_design.leg_area_ratio
length = te_design.length
R_ratio = te_design.R_ratio
I = te_design.I

SIZE = 100

R_desired_array = (
    np.linspace(0.5, 2., SIZE) * te_design.R_desired
    )

# important ones
I_array = (
    np.linspace(0.5, 2, SIZE) * I
    )
leg_area_ratio_array = (
    np.linspace(0.5, 2, SIZE) * leg_area_ratio
    )
R_ratio_array = (
    np.linspace(0.5, 2, SIZE) * R_ratio
    )
fill_fraction_array = (
    np.linspace(0.5, 2., SIZE) * fill_fraction
    )
length_array = (
    np.linspace(0.5, 2., SIZE) * length
    )

# Variables paired
```

```
power_R_fill = np.zeros(
    [R_desired_array.size, fill_fraction_array.size]
    )
power_fill_length = np.zeros(
    [fill_fraction_array.size, length_array.size]
    )
power_length_R = np.zeros(
    [length_array.size, R_desired_array.size]
    )

t0 = time.clock()
for index in np.ndindex(R_desired_array.size,
fill_fraction_array.size):
    i = index[0]
    j = index[1]
    if j == 0:
        print "i =", i

    te_design.R_desired = R_desired_array[i]
    te_design.fill_fraction = fill_fraction_array[j]
    te_design.set_constants()

    te_design.solve_te_pair_R()

    power_R_fill[i, j] = te_design.P_flux

te_design.R_desired = R_desired
te_design.fill_fraction = fill_fraction
te_design.length = length

t1 = time.clock() - t0
print "t1 =", t1
t0 = time.clock()
for index in np.ndindex(fill_fraction_array.size,
length_array.size):
    j = index[0]
    k = index[1]
    if k == 0:
        print "j =", j

    te_design.fill_fraction = fill_fraction_array[j]
    te_design.length = length_array[k]
    te_design.set_constants()

    te_design.solve_te_pair_R()

    power_fill_length[j, k] = te_design.P_flux

te_design.R_desired = R_desired
te_design.fill_fraction = fill_fraction
te_design.length = length
```

```
t2 = time.clock() - t0
print "t2 =", t2
for index in np.ndindex(length_array.size, R_desired_array.size):
    k = index[0]
    i = index[1]
    if i == 0:
        print "k =", k

    te_design.length = length_array[k]
    te_design.R_desired = R_desired_array[i]
    te_design.set_constants()

    te_design.solve_te_pair_R()

    power_length_R[k, i] = te_design.P_flux

te_design.R_desired = R_desired
te_design.fill_fraction = fill_fraction
te_design.length = length

data_dir = '../Output/te_design_space1/'
np.save(data_dir + 'power_R_fill', power_R_fill)
np.save(data_dir + 'power_fill_length', power_fill_length)
np.save(data_dir + 'power_length_R', power_length_R)
np.save(data_dir + 'R_desired_array', R_desired_array)
np.save(data_dir + 'R_ratio_array', R_ratio_array)
np.save(data_dir + 'fill_fraction_array', fill_fraction_array)
np.save(data_dir + 'length_array', length_array)
np.save(data_dir + 'I_array', I_array)
np.save(data_dir + 'leg_area_ratio_array', leg_area_ratio_array)

print "\nProgram finished."
print "\nPlotting..."

# execfile('plot_te_design_space_3dprojection.py')
```

Bibliography

1. U.S. Energy Information Administration, “Environment: Emissions of Greenhouse Gases,” Annual Energy Review 2009, Chapter 12, Table & Figure 12.1, 19 August 2010,.

2. M.L Parry, C Rosenzweig, A Iglesias, M Livermore, G Fischer, Effects of climate change on global food production under SRES emissions and socio-economic scenarios, Global Environmental Change, Volume 14, Issue 1, April 2004, Pages 53-67, ISSN 0959-3780

3. John D. Aber, Knute J., et al., Nitrogen Saturation in Northern Forest Ecosystems, BioScience, Vol. 39, No. 6, Jun 1989, pp. 378-386

4. Can Wang, Jining Chen, Ji Zou, Decomposition of energy-related CO2 emission in China: 1957–2000, Energy, Volume 30, Issue 1, January 2005, Pages 73-83, ISSN 0360-5442.

5. Jerry M. Melillo, John M. Reilly, et al., Science, Indirect Emissions from Biofuels: How Important?, Vol. 326, No. 5958, Dec 4 2009pp. 1397-1399.

6. A. Haines, R.S. Kovats, D. Campbell-Lendrum, C. Corvalan, Climate change and human health: Impacts, vulnerability and public health, Public Health, Volume 120, Issue 7, July 2006, Pages 585-596, ISSN 0033-3506.

7. U.S. Energy Information Administration, “Energy Explained: Coal – Coal and the Environment,” 08 July 2011.

8. U.S. Environmental Protection Agency, “Climate Change: Science – State of Knowledge,” 14 April 2011.

9. John Heywood. Internal Combustion Engine Fundamentals. McGraw-Hill Science/Engineering/Math, 1 edition, April 1988.

10. J. A Caton. Operating characteristics of a spark-ignition engine using the second law of thermodynamics: effects of speed and load. SAE Technical Paper, page 2000–01–0952, 2000.

11. T. Endo, S. Kawajiri, Y. Kojima, K. Takahashi, T. Baba, S. Ibaraki, T. Taka- hashi, and M. Shinohara. Study on maximizing exergy in automotive engines. Technical Report 2007-01-0257, SAE International, Warrendale, PA, April 2007.

12. Erik W Miller, Terry J Hendricks, and Richard B Peterson. Modeling energy recovery using thermoelectric conversion integrated with an organic rankine bottoming cycle. Journal of Electronic Materials, 38(7):1206–1213, March 2009.

13. W.M.S.R. Weerasinghe, R.K. Stobart, and S.M. Hounsham. Thermal efficiency improvement in high output diesel engines a comparison of a rankine cycle with turbo-compounding. Applied Thermal Engineering, 30(14–15):2253–2256, October 2010.

14. Hugues L. Talom and Asfaw Beyene. Heat recovery from automotive engine. Applied Thermal Engineering, 29(2–3):439–444, February 2009.

15. Waste heat recovery: Technology and Opportunities in U.S. Industry, US DOE, March 2008.

16. T. J Hendricks and J. A Lustbader. Advanced thermoelectric power system investigations for light-duty and heavy duty applications. II. In Thermoelectrics, page 387– 394. IEEE, August 2002.

17. Quazi E Hussain, David R Brigham, and Clay W Maranville. Thermoelectric exhaust heat recovery for hybrid vehicles. Technical Report 2009-01-1327, SAE International, Warrendale, PA, April 2009.

18. Douglas Crane, Gregory Jackson, and David Holloway. Towards optimization of automotive waste heat recovery using thermoelectrics. Technical Report 2001-01-1021, SAE International, Warrendale, PA, March 2001.

19. Hendricks, T.J., 2007. Thermal system interactions in optimizing advanced thermoelectric energy recovery systems. Journal of Energy Resources Technology 129, 223-231.

20. M. Gomez, R. Reid, B. Ohara, H. Lee. Influence of Electrical Current Variance and Thermal Resistances on Optimum Working Conditions and Geometry for Thermoelectric Energy Harvesting. Journal of Applied Physics, 113, 174908 (2013).

21. Recent Progress in Thermoelectric Power Generation Systems for Commercial Applications, 2011 MRS Spring Meeting. John W. LaGrandeur, Lon E. Bell, and Douglas T. Crane.

22. Y. Meydbray, R. Singh, Ali Shakouri, 2005 International Conference on Thermoelectrics (ICT), Thermoelectric-Module-Construction-for-Low-Temperature-Gradient-Power-Generation.

23. David Michael Rowe. Thermoelectrics Handbook: Macro to Nano. CRC/Taylor & Francis, Boca Raton, 2006.

24. Yang, Proceedings of 24th International Conference on Thermoelectrics (ICT) 2005, pp 170-174 (2005)

25. D.M. Rowe, International Journal of Innovations Energy Systems Power 1, 13 (2006).

26. D. M. Rowe, CRC Handbook of Thermoelectrics. (CRC Press, 1995), pp. 441-458

27. Richard Stobart and Dan Milner. The potential for thermoelectric regeneration of energy in vehicles. Technical Report 2009-01-1333, SAE International, Warrendale, PA, April 2009.

28. Richard K Stobart, Anusha Wijewardane, and Chris Allen. The potential for thermoelectric devices in passenger vehicle applications. Technical Report 2010-01-0833, SAE International, Warrendale, PA, April 2010.

29. Omer, S.A., Infield, D. G., 1998. Design Optimization of Thermoelectric Devices for Solar Power Generation. Solar Energy Materials and Solar Cells 53, 67-82

30. Xuan, X.C. 2002. Optimum Design of a Thermoelectric Design. Semiconductor Science and Technology. 17, 114-119.

31. Cheng, Yi-Hsiang, Lin, Wei-Keng. 2005. Geometric Optimization of Thermoelectric Coolers in a Confined Volume using Genetic Algorithms. Applied Thermal Engineering 25, 2983-2997

32. K. Matsubara, Proceedings of 21st International Conference on Thermoelectrics (ICT) 2002, pp. 418–423 (2002).

33. J. C. Bass, N. B. Elsner, F. A. Leavitt. Performance of the 1 kW Thermoelectric Generator for Diesel Engines. AIP Conference Proceedings 316, 295(1994).

34. A. S. Kushch, J. Bass, S. Ghamaty, N. Elsner. Thermoelectric Development at Hi-Z Technology. 20th International Conference on Thermoelectrics (2001).

35. Doug T. Crane. An introduction to system level steady-state and transient modeling and optimization of high power density thermoelectric generator devices made of segmented thermoelectric elements. Journal of Electronic Materials, 40(5):561–569, 2011.

36. Douglas T. Crane and Gregory S. Jackson. Optimization of cross flow heat exchangers for thermoelectric waste heat recovery. Energy Conversion and Management, 45(9–10):1565–1582, June 2004.

37. Kumar, Sumeet, et. al., 2013. Thermoelectric Generators for Automotive Waste Heat Recovery Systems Part I: Numerical Modeling and Baseline Model Analysis. Journal of Electronic Materials, Vol. 42, No. 4.

38. N. Espinoza, M. Lazard, L. Aixala, H. Scherrer. Modeling a Thermoelectric Generator Applied to Diesel Automotive Heat Recovery. Journal of Electronic Materials, v.39, no.9, 2010.

39. C. Baker. Simulation, Design, and Experimental Characterization of Catalytic and Thermoelectric Systems for Removing Emissions and Recovering Waste Energy from Engine Exhaust, The University of Texas at Austin, 2012.

40. Baker, C. and Shi, L., "Experimental and Modeling Study of a Heat Exchanger Concept for Thermoelectric Waste Heat Recovery from Diesel Exhaust," SAE Technical Paper 2012-01-0411, 2012, doi:10.4271/2012-01-0411.

41. C. Baker, P. Vuppuluri, L. Shi, and M. Hall, J. Electron. Mater. 41, 1290 (2012).

42. Chen, X; Weathers, A; Moore, A; Zhou, JS; Shi, L. Thermoelectric Properties of Cold-Pressed Higher Manganese Silicides for Waste Heat Recovery, Journal Of Electronic Materials,v.41,2012,p. 1564.

43. T. P Hogan and T. Shih. Modeling and characterization of power generation modules based on bulk materials. Boca Raton, FL: CRC Press, 2006.

44. Angrist, W. Angrist, 1977. Direct Energy Conversion, Carnegie-Mellon University

45. Adrian Bejan. Convection Heat Transfer. Wiley, 3 edition, July 2004.

46. Ling Shi, Gang Wu, et al. Interfacial Thermal Contact Resistance Between Aluminum Nitride and Copper at Cryogenic Temperatures. Heat and Mass Transfer (2012). 48:999-1004.

47. M. Yovanovich, J. R. Culham, P. Teertstra. Microelectronics Heat Transfer Laboratory, Department of Mechanical Engineering, University of Waterloo, 1997.

48. V. Satre, M. Lallemand. Enhancement of Thermal Contact Conductance for Electronic Systems, Applied Thermal Engineering 21 (2001) 221-235.

49. A. F Mills. Heat Transfer. Prentice Hall, 2 edition, August 1998.

50. http://www.electronics-cooling.com/1999/09/the-thermal-conductivity-of-ceramics/

51. http://global.kyocera.com/fcworld/charact/heat/thermalcond.html

52. http://www.surmet.com/technology/aln/index.php

53. R. Byron Bird, Warren E Stewart, and Edwin N Lightfoot. Transport Phenomena, 2nd Edition. Wiley, 2 edition, July 2001.

54. Michael J. Moran and Howard N. Shapiro. Fundamentals of Engineering Thermodynamics. Wiley, 5th edition, June 2003.